Improving Software Quality Assurance for Meter Data Management System: A Runtime Verification Approach

Matti Marttinen

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 21.9.2016

Thesis supervisor:

D.Sc. Seppo Sierla

Thesis advisor:

M.Sc (Econ) Antti Lammi

Aalto University
School of Electrical
Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Matti Marttinen

Title: Improving Software Quality Assurance for Meter Data Management System: A Runtime Verification Approach

Date: 21.9.2016 Language: English Number of pages: 7+65

Department of Automation and System Technology

Professorship: Information and Computer Systems in Automation

Supervisor: D.Sc. Seppo Sierla

Advisor: M.Sc (Econ) Antti Lammi

This thesis investigates business process oriented automatic testing for meter data management system. The purpose is to improve the quality assurance process for GENERIS meter data management system. The thesis identifies the most important processes of GENERIS meter data management system based on laws, regulations and guidelines. The most important of the identified processes are described as business process modelling notation diagrams that can be used for test design.

The thesis investigates feasibility of a new Quality Manager testing framework. The feasibility is analyzed by implementing a test case for market messaging process using the new framework. In addition, the feasibility of a virtual time management functionality for testing is analyzed. The new framework is also compared to an existing test automation tool. The effects of the new framework on a general software quality assurance process are analyzed. Case examples how the implemented test case improves the quality of the system are also presented.

It is established that the new framework is feasible for testing even though the test development consumes significantly more resources than with the old tool. On the other hand, the test scripts developed with the new framework require less maintenance and are more versatile.

Keywords: test automation, quality assurance, runtime verification

AALTO-YLIOPISTO
SÄHKÖTEKNIIKAN KORKEAKOULU

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Matti Marttinen

Työn nimi: Mittaustiedon hallintajärjestelmän laadunvarmistuksen parantaminen käyttäen ajonaikaista verifiointia

Päivämäärä: 21.9.2016 Kieli: Englanti Sivumäärä: 7+65

Automaatio- ja systeemitekniikan laitos

Professuuri: Automaation tietotekniikka

Työn valvoja: TkT Seppo Sierla

Työn ohjaaja: KTM Antti Lammi

Tässä opinnäytetyössä tutkitaan liiketoimintaprosessilähtöistä mittaustiedon hallintajärjestelmän automaattista testausta. Työn tarkoituksena on parantaa GENERIS-mittaustiedonhallintajärjestelmän laadunvarmistusprosessia. Työssä määritellään GENERIS-mittaustiedon hallintajärjestelmän tärkeimmän prosessit lakien, asetusten ja ohjeiden perusteella. Tärkeimmät tunnistetut prosessit on kuvattu bisnesprosessien mallinnuskaavioilla, joita voidaan käyttää testisuunnittelun pohjana.
Työssä tutkitaan uuden Quality Manager -testikehyksen soveltuvuutta. Soveltuvuutta tutkitaan toteuttamalla automaattinen testitapaus markkinaviestinnän prosessille. Lisäksi analysoidaan testikehyksessä olevan virtuaalisen ajan hallinnan soveltuvuutta testaukseen. Uutta testikehystä verrataan myös soveltuvin osin vanhempaan testiautomaatiotyökaluun. Uuden testikehyksen vaikutuksia yleiseen laadunvarmistusprosessiin analysoidaan. Lisäksi esitetään konkreettisia esimerkkejä, kuinka kehitetty testitapaus parantaa tuotteen laatua.
Testikehys näyttää olevan käyttökelpoinen työkalu, joskin testin kehittäminen vaatii huomattavasti enemmän resursseja vanhaan työkaluun verrattuna. Toisaalta uudella kehyksellä toteutetut testit vaativat vähemmän ylläpitoa ja ovat monipuolisempia.

Avainsanat: testiautomaatio, laadunvarmistus, ajonaikainen verifiointi

Preface

First, I want to thank my instructor Antti Lammi for the opportunity to write my thesis for Enoro and his valuable guidance in the process. Second, I want to thank everyone at Enoro who has made writing this thesis possible and who has supported me in the process. Especially, I thank Teemu Vättö for the technical insight he provided and Jaakko Korpi for his help and guidance regarding the QM. I also thank Mikko Lampinen and Ville Levänen for providing the test environment.

Third, I want to thank my supervisor, D.Sc Seppo Sierla, for his valuable input regarding the scope and content of my thesis. I always got swift and very helpful replies.

I would also like to thank all other people who have supported me during this summer and spring and my whole education. I thank my family for all the support I have received during these years in Aalto University and even before. Special thanks to Emmi Jeskanen for the understanding and patience when I felt desperate with this thesis.

Last but not the least, I would like to thank a group of friends who call themselves Nerdclub. We have shared the pains and gains during our education and have made it together. Now is the time to face new challenges, but I trust that our group will hold together in the future as well.

Leppävaara, 21.9.2016

Matti Marttinen

Contents

Abbreviations

AMR	automatic meter reading
AMI	advanced metering infrastructure
BPMN	business process modelling notation
DSO	distribution system operator
ISR	imbalance settlement responsible
MDM	meter data management
MDMS	meter data management system
QM	quality manager
QA	quality assurance
SQA	software quality assurance
SUT	system under testing
DST	daylight saving time
LTL	linear temporal logic
MTL	metric temporal logic

1 Introduction

As in all business, the quality of the product is of high importance in software industry. Quality products usually mean happier customers that can potentially lead to more sales. Quality is also a matter of profitability. According to Shull et al. [2002] the cost of finding and fixing a software defect after delivery can be 100 times more expensive than in the requirements and design phase of a project. Therefore it is crucial that as many defects as possible are discovered before delivering the product.

Because minimizing defects before delivering is important for software productivity the importance of testing and verification cannot be underestimated. Hailpern and Santhanam [2002] estimate that 50-75 % of software development costs comes from testing, debugging and verifying. Some estimates go even as high as 80% [Garousi and Zhi, 2013]. Therefore, minimizing these costs while maintaining quality would greatly improve software productivity.

A successful test automation could help to improve the productivity of a software by reducing the time and cost required for testing and increasing product quality [Rafi et al., 2012]. In test automation the mundane and time consuming tests are performed by another software. However, test automation is a large investment and requires expertise to implement. Like all testing, automated tests can be implemented on various levels of the software testing. It requires careful planning and evaluation to determine, which tests should be automated and which should be left for manual execution. The maintenance of test automation must also be taken into consideration.

In this thesis the focus is on automated verification of a meter data management (MDM) software called GENERIS. A new testing framework, Quality Manager (QM), has been developed within Enoro. The testing with QM framework is done using public interfaces of the system. A new approach that combines black-box testing and runtime verification is proposed. The goal of this thesis is to provide evidence that it is feasible to test business processes of GENERIS with the QM framework. In addition, feasibility of a time shifting functionality of the framework is examined.

First part of this thesis explores the background of quality management and software verification. In the scope of this thesis quality assurance is the most interesting part of quality management. Software quality assurance process is introduced as it is presented in IEEE 730 standard. Quality assurance aims to provide evidence about quality in which software testing plays a critical role. Fundamentals of software testing, test automation and runtime verification are presented to give background information about different testing approaches. Finally, meter data management systems are introduced using GENERIS as a case example as it is the target system for testing in this thesis.

After the background section, the research targets, materials and methods are

introduced. A lot of research is qualitative research. First the critical processes of an MDM system are identified and analyzed. The practical part of the research consists of implementing an automatic test case for market messaging. The implementation is used to demonstrate the feasibility of QM. Additionally, a comparison to an existing test automation system is presented. After the implementation, the effects on the general SQA process are analyzed. Additionally, bug findings are used to demonstrate the value of the implemented test case.

Finally, conclusions are drawn from the results and the thesis is discussed along with related work. Future development is also discussed.

2 Background

This section introduces the background for this thesis. The purpose of the section is to present the basic principles for quality management and software verification and provide motivation for the thesis. The section discusses standards related to quality management and quality assurance and fundamentals of software testing.

2.1 Fundamentals of Quality management

No matter what industry, quality is an important factor for revenue generation. Good quality products lead to better customer satisfaction and more opportunities for sales. It is therefore vital to monitor, assess and improve the quality. Software industry is no exception to this.

2.1.1 Basic concepts of quality management systems

Quality management is a broad concept. It can be defined for the whole business process of an organization as it is in the SFS-EN ISO 9000 standard. Quality management by definition means the actions taken in order to direct and control the quality of the product to a desired state. These actions within quality management may consist of all or some of the following:

- Quality planning
- Establising:
  - Quality policies
  - Quality objectives
  - Processes
- Quality assurance
- Quality control
- Quality improvement

[Suomen Standardoimisliitto SFS, 2015]

In the 1980s, the software industry developed and changed so rapidly that quality management systems couldn't effectively be adopted. This lead to poor quality products. For this reason the software quality management has been investigated extensively. This has lead to development of standards that are applicable to software industry, one of which is the ISO 9000 family. According to research by Tang and Chen [2010] the adoption of software quality management increases productivity, product quality, customer satisfaction and volume of business for most companies. [Tang and Chen, 2010]

Similar results were obtained by Hashmi et al. [2013]. According to Hashmi et al. [2013], the implementation of software quality standards clearly improves product development time, product safety and customer satisfaction. These are all important factors for the success of a software company, and it can be concluded that implementing a good quality management system is very beneficial for a software company.

SFS-EN ISO 9001 standard sets the requirements for a quality management system. The standard lists many potential benefits for an organization from implementing a system based on the standard. Particularly, the ability to provide products that meet the applicable statutory and regulatory requirements is of interest in this thesis. The ISO 9001 has been adopted specifically for computer software in the IEEE 90003. It provides guidelines how to apply a quality management system specified in ISO 9001 to a computer software framework [IEEE, 2015].

2.1.2 Software quality assurance process

The scope of this thesis is around quality assurance and automated software verification. The ISO 9001 standard mostly focuses on the management side of an organization. There are other standards that focus on the actual quality assurance process, such as IEEE 730 [IEEE, 2014]. The IEEE 730 specifies requirements for software quality assurance (SQA) process in software development or maintenance projects.

The IEEE 730 standard consists of overview, referenced normative standards, terms and abbreviations, key concepts and the specification for actual software quality assurance process. One of the key concepts of software quality assurance process is the organizational management responsibility. Minimally it means that management needs to understand the SQA process, provide the resources for it and act based on the information that SQA provides. [IEEE, 2014]

Another key concept for SQA is the relationship between the project and the organization. The organization is responsible for establishing the SQA framework, overseeing the project and mostly negotiating the contract. The project can take part in negotiating the contract as well. As for the project, it is responsible for SQA planning, SQA activities and SQA closing. Project-wise, the role of SQA ends with the project closing. The project is closed after the release of the product and acceptance of the project. The established SQA framework, however, may be preserved for future projects as well. [IEEE, 2014]

Another important concept is the software quality and its relation to the requirements. Quality is the product's ability to meet the expressed or implied requirements set by the stakeholders. These stakeholder requirements are refined into functional and performance requirements of the software. The SQA's purpose is to provide evidence that the produced software meets these requirements. [IEEE, 2014]

The software quality assurance process can be divided into subsets of activities. They are categorized in the IEEE 730 standard as follows:

- SQA process implementation activities
  - Establish the SQA processes
  - Coordinate with related software processes
  - Document SQA planning
  - Execute the SQA plan
  - Manage SQA records
  - Evaluate organizational independence and objectivity
- Product assurance activities
  - Evaluate plans for conformance to contracts, standards, and regulations
  - Evaluate product for conformance to established requirements
  - Evaluate product for acceptability
  - Evaluate product life cycle support for conformance
  - Measure products
- Process assurance activities
  - Evaluate life cycle processes and plans for conformance
  - Evaluate environments for conformance
  - Evaluate subcontractor processes for conformance
  - Measure processes
  - Assess staff skill and knowledge

[IEEE, 2014]

Agile software development methodologies might utilize an SQA process that differs from the one in IEEE 730. Hongying and Cheng [2011] propose an agile quality assurance model (AQAM) that can be used by small and medium sized agile software development teams. AQAM is divided into key process areas (KPAs) that provide the guidelines, benefits, processes, templates, customization and maturity levels. [Hongying and Cheng, 2011]

KPAs of the AQAM include requirement management, process audit, test planning, test case management, peer review, defect analysis, defect reporting, unit testing, performance testing, configuration management, management support, training, test environment management, test organization, test automation, continuous integration, user experience management, testing level, defect prevention and static analysis. The KPAs of the AQAM are not equally important and development teams are

encouraged to choose and focus on KPAs according to their needs. [Hongying and Cheng, 2011]

The AQAM process starts with evaluating current QA maturity. After this, the determined business goal is converted to a QA maturity goal. Next steps are selecting QA maturity KPA and developing an implementation proposal. The proposal plan is then implemented and the results are evaluated in the final step. The process can then start over with a different business goal. [Hongying and Cheng, 2011]

The AQAM is similar yet different to the SQA process described in IEEE 730 standard. The KPAs reflect the activities in the standardized SQA process. The IEEE 730 defines purpose for each activity, while the AQAM equivalent are the guidelines. The AQAM lists benefits for each KPA while the standard process lists outcomes. The processes in the AQAM are comparable to the tasks in the standardized SQA process. The AQAM KPAs feature best practises, which are absent from the standardized process.

Another approach is presented by Saif et al. [2010]. They present a framework that focuses on QA planning. The steps in their framework are determining influence factors, determining SQA strategy, evaluating SQA strategy, selecting most fitting SQA strategy, fine tuning QA techniques and measuring and analyzing the effects of the QA techniques. [Saif et al., 2010] In other words, their framework focuses on SQA implementation activities and has features from process assurance as well.

These QA models have similarities with the standardized SQA process. The following subsections focus on describing the standardized SQA process in order to understand the underlying organization level process that drives all verification activities.

While the utilization actual Quality Manager framework that is under the inspection in this thesis is used for product assurance activities the implementation of the QM framework and the test cases fall under SQA process implementation activities. The QM framework enables production environment-like simulation of business processes and it is used in product assurance. The framework will be introduced in more detail in section 4.2.1.

The SQA process implementation activities and product assurance activities are discussed in more detail in the following sections. The process assurance activities are mostly out of the scope so they will be only briefly introduced after product assurance activities. However, the product quality is dependent on the process quality, so the process assurance mustn't be neglected [Jae Won Lee et al., 2005].

2.1.3 SQA process implementation

Most of the SQA process implementation is done by the management side of the company. First activity is establishing the SQA processes. These processes should exist separately from the projects and they define the role, concepts, methods, procedures and practices for SQA function. [IEEE, 2014]

Another activity related to SQA process implementation is coordination with other processes [IEEE, 2014]. Redundant tasks should be minimized, because redundancy is highly inefficient use of resources. This makes it important to coordinate the QM implementation activities with existing verification and validation processes. The next activity in the list is documenting the SQA plan which aims to identify and document the project-specific SQA activities [IEEE, 2014].

After documenting the SQA plan comes the plan execution. Within this activity SQA tasks that are defined in the plan are executed and the SQA reports are created in order to evaluate software quality. The product and process non-conformances are raised if the expectations are not met. The next activity in the list, managing the SQA records, makes sure that the records of SQA activities, outcomes and tasks are created and available to project stakeholders. [IEEE, 2014]

The last activity listed under SQA process implementation is evaluating the organizational independence. The goal of this activity is to determine whether the persons that are responsible for SQA have sufficient authority and resources to evaluate the quality objectively and to tackle the problems that they find. They must also be able to communicate freely with the organization management. [IEEE, 2014]

The KPAs of the AQAM are not directly comparable to the activities of the IEEE 730 standard SQA process. However, management support, test organization, test planning and defect reporting are the KPAs that would be most close to the SQA process implementation activities. Reporting is listed in the standard as well as in AQAM KPAs. Management support is also closely related to the evaluating the organizational independence as the objective of the activity was to determine that the SQA function has sufficient authority and resources.

This concludes the introduction of the SQA process implementation activities. The implementation of QM framework and QM tests can be categorized under the SQA process establishment activities. Out of the SQA process implementation activities, this thesis is most relevant to SQA planning and coordinating with other processes. The next section discusses the product assurance activities.

2.1.4 Product assurance

Product assurance activities are the activities that actually provide confidence about the product quality. They can be further divided into: evaluating plans for conformance, evaluating product for conformance, evaluating product for acceptability, evaluating product lifecycle for conformance, and measuring products. The fundamental purpose of these product assurance activities is to prove that all the software services, products and related documentation comply with the contract. After these activities, all the non-conformances should also be addressed. [IEEE, 2014]

The IEEE 730 standard defines the outcomes and tasks for all of the product assurance activities. The goal of evaluating plans is to make sure that the plans conform to the contract, and that the contract is compatible with legislation. Also, the plans must

be documented and consistent with one another. [IEEE, 2014] The *test planning* KPA can also be related to these plan evaluation activities, such as.

Evaluating the product for conformance to the requirements is an activity where non-conformances are raised by the software product itself or the related documentation. Software validation, verification and review is conducted within this activity. [IEEE, 2014] Therefore, it can be argued that this is the most important activity for quality assurance.

Evaluating product for acceptance is a similar activity. With this activity, the supplier should confirm that the requirements are acceptably fulfilled. The acquirer might also verify that the product is acceptable. The activity is done before delivery and all the non-conformances are raised. The acceptance criteria are identified and documented and the product is tested against these criteria.[IEEE, 2014] Evaluating the product against requirements and for acceptance can be connected to multiple KPAs of the AQAM, such as unit testing, performance testing.

Evaluating the life-cycle support for conformance is also part of product assurance. The purpose of this activity is to verify that the planned support requirements are consistent with the contract and that the supplier and acquirer roles are clearly defined. Final activity related to product assurance is measuring the products. With this activity, the organization determines that the product measurements reflect the quality and that they conform to the standards and procedures utilized in the project. [IEEE, 2014]

The third part of SQA process is the process assurance. As the name implies the purpose of process assurance activities is to make sure that the development, operation, installation and maintenance processes comply with regulations and are able to consistently produce software products that meet the requirements. The process assurance activities consist of evaluating life-cycle processes, environments, and subcontractor processes, measuring the processes and assessing staff skills and knowledge. [IEEE, 2014] Many of the KPAs of the AQAM deal with the process assurance. For example, test environment and process audits are directly related to the process assurance activities.

This section gave an overview of quality management and the software quality assurance process. The next section discusses the fundamentals of software verification concepts and methods. Software verification is an important part of SQA, since the product assurance activities must provide proof that the product meets the requirements.

2.2 Software Verification: Basic Principles and Methods

According to IEEE standard 1012-2012 verification means all activities taken to provide evidence that the system, software or hardware and any associated products meet the requirements set for it [IEEE, 2012]. This section discusses the traditional

verification approaches. Software testing is a common way to verify the software systems against the requirements. This thesis utilizes an approach called runtime verification which is a lightweight verification technique that shares similarities with oracle based testing [Leucker and Schallhart, 2009].

2.2.1 Software testing

Software testing can be defined as dynamic verification of a program against the expected behaviour [ISO/IEC/IEEE, 2010]. The expected behaviour emerges from the requirements set for the software. Requirements can be be defined in collaboration with a customer, or they can emerge from general needs for the software. A combination of these sources is also possible. Software projects can be very different in nature. They can vary from off-the-shelf applications intended for retail customers to immense data systems tailored for a specific company with very detailed needs. Naturally, this affects the way the requirements are formed.

Multiple testing stages, also called levels or phases, can be identified for the software testing process. There is no definite scheme for testing stages. However, a typical layout that is based on classical V-model can be used as a base for standardized testing levels. Depending on the software development model used, the phase set might be different. [Majchrzak, 2012]

First level in the V-model based testing levels is called unit testing [Majchrzak, 2012]. Unit testing is defined as testing an individual software unit [ISO/IEC/IEEE, 2010]. On this stage developers or independent testers test single modules made by software developers. Logical errors in the coding should get caught on this level. After this stage there is an optional testing stage, where central components are tested on a test system, rather than developers' stations [Majchrzak, 2012]. Units or components can mean either procedures and functions, classes and methods, or some other small reusable components of the system. [Burnstein, 2003] In GENERIS there all of these are possible targets for unit testing.

The next level is called integration testing. At this stage, individual modules developed by possibly different developers are tested against each other. [Majchrzak, 2012] The goal of integration testing is to detect problems in the interfaces between the units and to assemble the units into working subsystems and eventually into a complete system [Burnstein, 2003]. If problems in interoperability are discovered later on, this stage must be repeated. After the integration test there is an optional performance test stage. [Majchrzak, 2012]

Following the integration and optional performance test stages is the system testing level. This is a large scale integration test for the whole system. It is conducted by black-box testing methods and is less technical than the previous levels. In the previous stages some components may have been simulated by so called *test stubs*. At this stage such simulations are removed and the system is fully integrated. Typically, the testers on this level are not the developers who made the code but other people

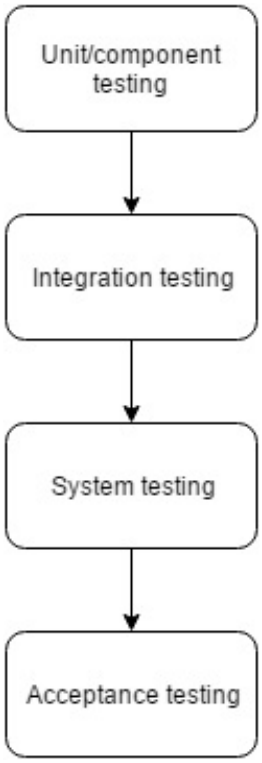


Figure 1: Typical testing levels, based on [Majchrzak, 2012]

with no knowledge of the source code. [Majchrzak, 2012] System testing can consist of functional testing, performance testing, stress testing, configuration testing, security testing and recovery testing [Burnstein, 2003].

Next step in the testing scheme is the acceptance testing. In agile software development acceptance tests are regarded as key tests. [Majchrzak, 2012] At this stage the customer or the user verifies that the system meets the specifications set for it [IEEE, 2012]. After customer acceptance, the software is taken to pilot test in which it is tested on multiple systems and can eventually be taken into product use [Majchrzak, 2012].

The division of testing levels described above is a classical one and it is also described by Burnstein [2003]. The levels comply with the IEEE Standard for System and Software Verification and Validation. In the standard the levels are named differently but are ultimately the same. The levels in the standard are component testing, integration testing, qualification testing and system testing [IEEE, 2012]. Figure 1 illustrates the the testing levels.

In addition to testing levels, software testing methods can be divided into three groups: static, dynamic and exhaustive methods. Exhaustive testing only works for the simplest of programs and is therefore meaningless in the scope of large software products. [Majchrzak, 2012]

Static methods don't involve executing the actual program. They can be further

split into verification and static analysis. In verification, program is formally proved to be correct. As for static analysis, it evaluates components or systems based on their form, structure and documentation. [ISO/IEC/IEEE, 2010] Static methods only give hints about defects in software [Majchrzak, 2012].

Dynamic methods are the opposite and their purpose is to find defects by executing programs. A useful classification for the dynamic methods can be derived from the availability of information for the test case. If the test is done without information about the code, it is called black-box testing. On the contrary, if the test utilizes the source code it is called white-box (or glass-box) testing. The mixture of the two is called gray-box testing. In gray-box testing the source code is not necessarily utilized but the testing is done by people that know the source code. [Majchrzak, 2012] According to Khan and Sadiq [2011] the black-box testing requires fewer resources. This makes sense since in the case of black-box testing the people who know the source code can be allocated to development tasks in stead of testing.

Dynamic methods can be further categorized to two main categories: structural and functional methods. Structure based methods assess the program based on data and execution flows. Therefore they are white-box tests. Functional tests are based on test cases that are derived from the specification. Functional tests are often black-box tests. [Majchrzak, 2012] This thesis' runtime verification approach shares similarities with functional black-box tests so they are specified further in the following.

Functionality oriented testing can also be referred to as specification oriented testing, because the testers determine the test cases based on specifications. With function oriented testing one doesn't necessarily reach full test coverage for the software but it helps to ensure that specifications are met. One of the simplest function oriented methods is testing for function coverage which means that every function in the program is tested with one test. This method is inefficient and faces severe limitations. [Majchrzak, 2012] It might be more efficient to leave simplest function tests to static verification.

Use case means a complete task of a system [ISO/IEC/IEEE, 2010]. In other words, use cases tell how the software is actually used. A form of scenario testing, use case testing, uses use case models to describe the sequences of interactions between the test item and other actors. These sequences are tested based on the models. [ISO/IEC/IEEE, 2015] Use cases may have different paths inside the program and for each use case as many tests must be performed as is needed to cover all the paths [Majchrzak, 2012].

Use case oriented testing is arguably the most useful testing method, as the end users operate the software according to the use cases.

In addition, functionality oriented black-box testing methods include *equivalence partitioning* (or equivalence class partitioning) and *boundary value analysis* [Majchrzak, 2012, Khan and Sadiq, 2011]. The equivalence partitioning requires the system under testing (SUT) to have well defined input and outputs [Burnstein, 2003]. The input domain can then be divided into partitions, for example, valid and invalid inputs. It

is then assumed that each partition or class behaves in a reasonably same manner [ISO/IEC/IEEE, 2015], so it is sufficient to test each partition once. Therefore, the equivalence partitioning eliminates the need for infeasible exhaustive testing [Burnstein, 2003].

The boundary value analysis utilizes the equivalence classes from equivalence partitioning but the testing focuses on the boundaries of the classes. [Burnstein, 2003] Naturally, this requires the classes to have clear boundaries. For this reason the boundary value analysis is best suited for situations where SUT has numerical input data.

If the system is a finite-state machine, a useful testing approach that can be used is *state transition testing* (or state oriented testing [Majchrzak, 2012]). The SUT is considered as a set of states and transitions between the states. State transition testing may reveal defects that can't be detected with input/output based methods. A state transition graph (STG) is a graph that shows all the possible states as nodes and directed edges represent transitions between states. [Burnstein, 2003]

With STG one can identify which execution paths are allowed. However, if the amount of transitions is high the state transition testing becomes inapplicable [Majchrzak, 2012]. In this type of testing the state information must be publicly available if black-box testing is used. One way to give information about the states could be writing logs that can be used to identify states and transitions.

There are more testing methods available in literature, but considering the scope of this thesis they are not introduced here. The testing methods that were introduced above are the most important for the scope of this thesis. They are all black-box tests, and this thesis concentrates black-box verification of an MDM system.

2.2.2 Test automation

This section discusses the basic concepts of test automation. An overview of the basic methods, advantages and challenges of test automation is presented.

Test automation means more than just running test cases automatically although the automatic test execution is the most popular domain of test automation. The test automation categories are test management, unit test, test data generation, performance test and test execution. The test execution automation has the best return on investment (ROI) opportunity which explains why it is the most popular domain. [Wissink and Amaro, 2006]

Even though test automation is seems like the most natural application field for automation, the test automation projects are risky [Wissink and Amaro, 2006]. The test strategy is vital for the success of test automation. The test automation strategy should define which levels of testing are included in the test automation. It should also be defined whether automation is used for functional, performance or reliability

tests. [Berner et al., 2005] Berner et al. list four major faults for test automation strategies:

- Misplaced or forgotten test types
- Wrong expectations
- Missing diversification
- Tool usage limited to test execution

[Berner et al., 2005]

Misplaced or forgotten test types can easily lead to inefficient testing if, for example, unit testing is being carried out on system level. On system level it is difficult or impossible to force the SUT into the states that are required for unit tests. Berner et al. also say that many organizations tend to neglect difficult tests such as robustness tests. [Berner et al., 2005]

Unrealistic expectations can also be problematic for a test automation system. If too high a ROI is expected but not achieved the test automation can easily be abandoned. When calculating the ROI for test automation, the shorter release cycle and the improvement of testing quality should also be considered. The quality improvement is caused by the fact that testers have more time to design better tests when the time is not spent on actual testing. [Berner et al., 2005]

A good test strategy should combine unit, integration and system test automation. Another thing to consider is the tool usage outside test execution. Tools can be used for designing test cases and reports, analysis and reporting. Additionally, a good test management tool can save even more resources than automating test execution. [Berner et al., 2005]

When automating test cases, it is very important to consider the repetition of the test. The cost-effectiveness depends highly on the times the automated test is executed [Berner et al., 2005]. Test automation systems and automated test cases naturally need a lot of expertise to develop and maintain. Therefore it is not feasible to automate tests that are not run often enough.

However, the amount of test executions to make test case automation feasible is surprisingly low. If the test case is expected to run ten times, it would be potentially beneficial to automate it. The execution count is seldom the reason for failing test automation systems. The fault usually lies within the testing strategy or application architecture that doesn't support testing very well. [Berner et al., 2005]

Persson and Yilmazturk [2004] list the underestimation of human intervention for automated testing as a common pitfall for test automation. This complies with Berner et al. [2005] as they also state that the capability to run automated tests reduces if the tests are not run often enough [Berner et al., 2005]. This results from the fact that the tests need maintenance in order to keep up with the development of the SUT.

An important domain of test automation is the test oracle problem. The term test oracle refers to the entity that determines whether the test passed or failed [Li and Offutt, 2014]. More complex tests require more complex test oracles. This makes developing a complex test automation system as difficult as it is. Without solving the test oracle problem, the test automation system will not be efficient as humans have to make the verdict about the test, even though the execution might be automatic. The automated test oracle process has three steps: generation of expected outputs, comparing the expected output with actual output from the SUT and detecting the abnormalities [Vineeta et al., 2014]. The process is shown in figure 2.

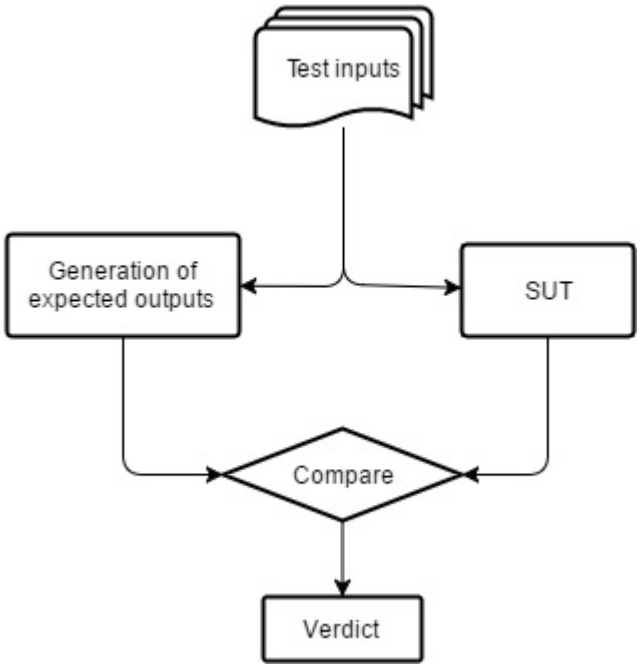


Figure 2: A general automatic test oracle process, based on [Vineeta et al., 2014]

In regression testing, the development of the automatic test oracle can be easier. Regression testing implies that the previous version is correct, so it could be used to generate expected outputs [Harman et al., 2013]. In this case, no difficult development for an oracle is needed since the previous version can be used. However, the previous version must be run parallel with SUT.

2.2.3 Runtime verification

Runtime verification is another approach to software verification that has gained a lot of attention from the researchers and practitioners during last decade [Falcone and Zuck, 2015]. Runtime verification is used to dynamically verify the behaviour of a system run against given requirements [Colin and Mariani, 2005]. It is often used to complement other verification methods such as testing, model checking and theorem proving. [Leucker and Schallhart, 2009]

A*monitor* is an entity that observes the execution of the program against certain correctness properties and decides whether the execution run of the system violates these properties. The correctness properties, or requirements, are usually formulated in some variant of linear temporal logic (LTL). [Leucker and Schallhart, 2009]

The variant used in this thesis is *metric temporal logic (MTL).* This variant takes into account the metric nature of time. MTL is introduced originally by Alur and Henzinger [1990]. They use notation $\Box(p \rightarrow \Diamond_{leq5} q)$ to formalize statements such as "Every p-state is followed by a q-state within time 5" [Alur and Henzinger, 1990].

In addition to such statements, this thesis has a need for statements like: *Every p-state is followed by a q-state after time* t_1 *but before time* t_2. This can be achieved by applying a lower bound to the temporal requirement. Thati and Roşu [2005] use notation $\Diamond_I$, where I can be an interval of non-negative real line. If only interval $[0, \infty)$ is allowed, MTL becomes pure LTL [Thati and Roşu, 2005].

By equipping the final product with a monitor, it would be possible to detect violations and react to those. Ability to react and recover from violations is unique to runtime verification. [Leucker and Schallhart, 2009] For now, the violation recovery and reacting is out of the scope of the QM and therefore out of the scope of this thesis. The QM is used solely to verify the correct behaviour. The function of the monitor is essentially the same as test oracle's.

Figure 3 shows an high-level description of monitors. Monitors are derived from the requirements and they monitor the system execution. An event handler, if implemented, can be used to recover from the violations of correctness properties.

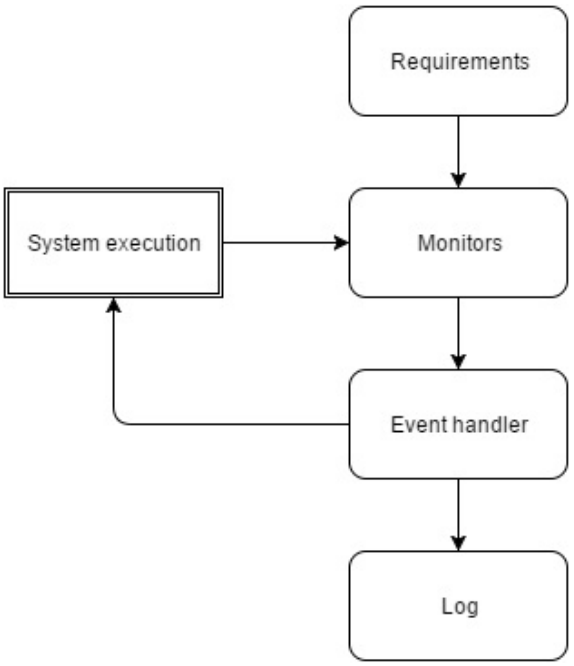


Figure 3: Monitors in runtime verification, based on [Delgado et al., 2004]

There are numerous ways to implement monitors and Delgado et al. [2004] describe a way to classify the monitors. Their division is based on four features:

- Monitoring points
- Placement
- Platform
- Implementation

Monitoring points are the points in the program where the monitoring code is executed. The classification is based on how the points are derived. The placement feature describes where the monitoring code is executed. It can be executed *inline*, in other words within the target code, or *offline*, meaning the monitor executes in a different process. [Delgado et al., 2004]

Platform division is used to classify the monitors between software and hardware monitors. Implementation feature further classifies the placement feature. The monitor can be implemented as a *single process* which corresponds to the inline placement. The monitor is then executed in the same process with the target program. *Multiprogramming* implementation means that the monitor is executed as a separate process whereas *multiprocessor* implementation means that the monitor is executed by a different processor than the target program. These both are subclasses of offline placement. [Delgado et al., 2004]

Runtime verification can also be interpreted as oracle based testing. However, in testing the oracle is often defined directly rather than being derived from high-end specifications. Another difference to testing is that the runtime verification seldom considers exhaustive input sequences to the system. Runtime verification does not influence the program's execution in any way even when it detects a violation of expected behaviour. [Leucker and Schallhart, 2009]

The difference between an oracle in testing and a monitor in runtime verification can be obscure. In fact, Colin and Mariani [2005] state that the difference is that the oracle is used for verification in the testing phase but a monitor is used after deployment. The implication is that the two terms represent the same function.

2.3 Meter Data Management Systems: Overview and Interfaces

Smart grid is the future of the electricity distribution system. According to Energiateollisuus [2016], Finland was the first country to almost fully utilize hourly based electricity measurements. That is undeniably the first step towards full smart grid utilization.

With the smart grid utilization, the amount of metering data that is being communicated between the meters and utilities will increase significantly. This makes a meter

data management system (MDMS) a key requirement for smart grid infrastructure. The MDMS is used to store, manage and analyze the metering data. [Gungor et al., 2013]. For example, the hourly interval in electricity metering means that there will be there will be 24 data points each day of the year, for every hourly measured metering point Finland. The metering point register governed by Fingrid includes approximately 3.4 million metering points [Fingrid, 2016]. This would mean 81.6 million measurements a day. Finland has many distribution system operators (DSO), so not all of these measurements go through a single MDMS. However, in bigger countries and bigger DSOs this might be a reality.

An example MDM system was developed in [Matheson et al., 2004]. Their system is based on the best practices set by Edison Electric Institute. The situation is comparable to Finland, where the recommendations are given by Energiateollisuus Ry.

This thesis focuses on MDMS called GENERIS and the next sections describe it in more detail.

2.3.1 GENERIS

GENERIS is a MDM system from Enoro. It is intended for all parties that operate in the Finnish electricity markets: distribution system operators (DSO), retailer (RE), balance responsible parties (BRP) and transmission system operator (TSO). It also includes functionalities for district heating and gas. The roles have inherently different processes and functions, as the responsibilities of these market parties are different. The focus of this thesis is in the DSO scope of GENERIS although QM will ultimately be used to perform system level verification for all scopes.

The GENERIS system runs on top of an Oracle database. All the data is stored in the database. This includes the time series data and the master data. Time series data contains the measurement data from metering points. Time series data includes the statuses for the values in the time series. Master data contains general information about the metering points such as customer, contract, supplier and grid connection information. Master data also includes the links to the related time series data.

Since GENERIS has capabilities for so many different functions it is most accurately described as a platform. The features of the platform are utilized to create a product that meets the utility's needs. The core of the GENERIS platform is Info Flow Manager (IFM) that coordinates the scheduling and information flow of the processes, also known as IFM jobs, within GENERIS. These jobs run the automated processes needed in the MDM system.

2.3.2 Interfaces

An MDM system requires multiple interfaces in order to accomplish the necessary functionalities. Jung et al. [2012] list following channels for data exchange in the smart grid:

1. Utility to Customer
2. Utility to Utility
3. Internal Exchange
4. Smart Meter to Advanced Metering Infrastructure (AMI): IP gateway or in-home display
5. IP Gateway to Utility or Customer

This section discusses the interfaces on a general level for any MDM system and gives examples of these interfaces in GENERIS. An overview of the interfaces is shown in figure 4.

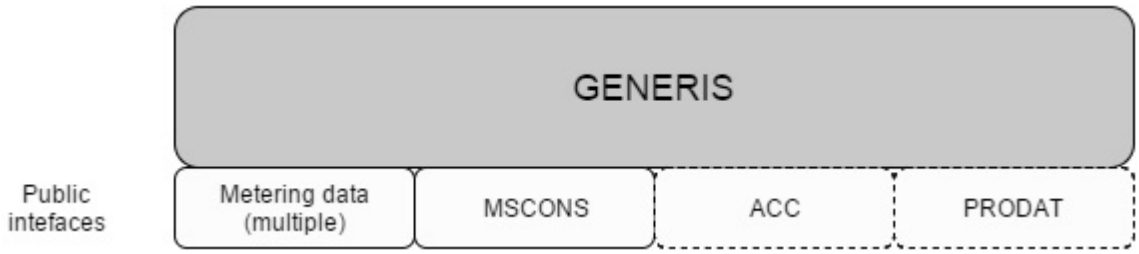


Figure 4: Public interfaces of GENERIS

First of all, an MDM system requires an interface for time series data. This is related to the data exchange number 4 in the division by Jung et al. [2012]. An additional exchange channel, *AMI to Utility* could be used to make the list more complete, as DSOs are allowed to outsource the metering [Työ- ja Elinkeinoministeriö, 2013]. Time series data is required in every critical business process in DSO scope. No matter which MDM software, it needs to be able to receive time series data.

The GENERIS system has multiple interfaces for time series data. Typically the time series data interfaces work through the file host system. Using the file interface is not the only way, but the other options are left outside the scope of this thesis. The GENERIS system polls a certain directory for a certain type of file and when a file is found the system imports the data using interface specific rules.

This thesis uses a standard ascii file (SAF) interface for time series data. SAF interface is a simple GENERIS specific interface. The SAF interface was chosen because QM framework supports only SAF file generation at this point.

Another important interface is the interface for outgoing messages. In this case, it means utility to utility communication in the division by Jung et al. [2012]. An

MDM system needs to be able to create messages for market parties. In Finland the messages sent from the DSO to retailer are called metered services consumption report (MSCONS) messages and are based on the UN/EDIFACT standard [Fingrid, 2015]. The standard is internationally used for pricing information in smart grid systems [Tariq et al., 2012].

In the future, there will also be interfaces for ebIX and ENTSO-E messages that are used in the message flows between the DSOs and the imbalance settlement responsible (ISR) party [eSett, 2016]. These interfaces are taken to use when the Nordic Imbalance Settlement is launched in Finland, Sweden and Norway. However, the ebIX and ENTSO-E interfaces are not considered in this thesis, as they are not yet required from an MDM system.

An example MSCONS message from imaginary DSO MMR to imaginary supplier SUPP is presented in figure 5. The UNB segment contains information about the message itself, such as syntax version and sender and recipient EDIEL identification codes. DTM segments are used to describe time stamps. The number after the '+' that follows the code DTM is used to determine the use and format of the time stamp. NAD segment describes the parties related to the message. The FR specifier is used for sender while DO is used for the document recipient. LOC segment is used to identify the time series to which the following consumption values are related to. The value is presented in the QTY segment. The QTY segment has the status for the measurement and the consumption value with negative sign.[Ediel, 2002, 2005]

Oftentimes an electricity utility might have a separate customer information system (CIS) from the MDM system. In such a case an interface between the customer information system and MDM system is required. This represents the internal exchange communication in [Jung et al., 2012].

In GENERIS, this interface is called advanced CIS communication (ACC). The working principle is the same as for time series data. A specific type of file needs to be in the polling folder and the GENERIS system imports the data from that file and saves it to the database. This interface is GENERIS specific and it is not based on any standard. The ACC works with basic ASCII text files that have a specific structure.

The ACC interface might not be required for all MDM systems, as it is possible that the customer information system is an integrated part of the MDM system. In that case the system also needs an interface for PRODAT messaging. PRODATs are used for messaging market parties about changes in contracts and metering point information [Energiateollisuus Ry, 2013]. As the focus of this thesis is on meter data management, the interfaces for master data are left for little discussion.

```
UNA:+.? '
UNB+UNOB:2+MMR:ZZ+SUPP:ZZ+160729:1718+45'
UNH+1+MSCONS:D:96A:ZZ:E2FI02'
BGM+7+45+9+AB'
DTM+137:201607291718:203'
DTM+163:201607280000:203'
DTM+164:201607290000:203'
DTM+ZZZ:3:805'
NAD+FR+MMR:160:SLY'
NAD+DO+SUPP:160:SLY'
UNS+D'
NAD+XX'
LOC+90+FI_SUPP_MMR_MMRTEST0001::SLY+::SLY:MMR+SUPP::SLY'
RFF+LI:45_1'
LIN+1++1008:::SLY'
MEA+AAZ++Z02'
QTY+136:-1.182'
DTM+324:201607280000201607280100:Z13'
QTY+136:-3.902'
DTM+324:201607280100201607280200:Z13'
QTY+136:-3.613'
DTM+324:201607280200201607280300:Z13'
QTY+136:-4.832'
DTM+324:201607280300201607280400:Z13'
QTY+136:-3.624'
DTM+324:201607280400201607280500:Z13'
QTY+136:-0.601'
DTM+324:201607280500201607280600:Z13'
QTY+136:-0.877'
DTM+324:201607280600201607280700:Z13'
QTY+136:-4.793'
DTM+324:201607280700201607280800:Z13'
QTY+136:-1.496'
DTM+324:201607280800201607280900:Z13'
QTY+136:-3.864'
DTM+324:201607280900201607281000:Z13'
QTY+136:-3.594'
DTM+324:201607281000201607281100:Z13'
QTY+136:-4.986'
DTM+324:201607281100201607281200:Z13'
QTY+136:-3.675'
DTM+324:201607281200201607281300:Z13'
QTY+136:-3.645'
DTM+324:201607281300201607281400:Z13'
QTY+136:-3.919'
DTM+324:201607281400201607281500:Z13'
QTY+136:-2.142'
DTM+324:201607281500201607281600:Z13'
QTY+136:-2.395'
DTM+324:201607281600201607281700:Z13'
QTY+136:-4.196'
DTM+324:201607281700201607281800:Z13'
QTY+136:-2.209'
DTM+324:201607281800201607281900:Z13'
QTY+136:-3.178'
DTM+324:201607281900201607282000:Z13'
QTY+136:-1.405'
DTM+324:201607282000201607282100:Z13'
QTY+136:-1.695'
DTM+324:201607282100201607282200:Z13'
QTY+136:-1.144'
DTM+324:201607282200201607282300:Z13'
QTY+136:-3.013'
DTM+324:201607282300201607290000:Z13'
CNT+1:-69.980'
UNT+64+1'
UNZ+1+45'
```

Figure 5: MSCONS message example

3 Related work

This section discusses the related work to this thesis. The papers related to MDM software research were sparse and none of them discussed software quality. For this reason, papers that generally discuss software quality assurance, software verification and test automation were reviewed.

Garousi and Felderer [2016] write on developing and maintaining test automation scripts. This is directly related to the practical part of this thesis since a test script was developed as a part of the thesis. Garousi and Felderer call the process of developing and maintaining test scripts *software test code engineering* (STCE). According to the paper, the test script development is tedious, prone to error and requires large investments.[Garousi and Felderer, 2016] This complies with the findings of this thesis. The paper mentions three problematic types of tests that can lead to bad quality in test scripts: eager test, conditional logic and assertionless test. An eager test tries to verify too many functionalities and becomes difficult to understand and maintain. A conditional logic type test has multiple control flow paths which can make it difficult to see which parts of the test are executed. An assertionless test is a test that doesn't actually assert anything. Instead it only pretends to do so. [Garousi and Felderer, 2016]

In order to produce good quality test scripts for the QM these kinds of tests should be avoided. Eagerness is avoided in this thesis by creating a test case for a single business process, market messaging. Potentionally, a single test script could be used to verify other processes as well, such as balance settlement and supplier changes. This would make the test much more interesting, as the market messaging process without supplier changes is quite constant for the whole year. However, such modifications would make the test much more eager. Eagerness is an unwanted property. Therefore, this contradiction needs consideration. One option is to check only interesting times, such as daylight saving time changes and leap years.

The state flows within the test are rather linear and lead to single exit points as can be seen from the UML diagrams describing test implementation in section 5.2.1. Therefore conditional logic has been rather well avoided.

The oracle problem is something to be considered when developing automatic software testing. Barr et al. [2015] address this problem with a survey to current approaches to the test oracle problem. They list the following types of test oracles: specified test oracles, derived test oracles, implicit test oracles and human test oracles. Human test oracles naturally mean that humans do the verdict about the test. Specified test oracles are derived from specifications. Derived test oracles are based on information received from documentation, system executions or other properties of the SUT. Implicit test oracles determine the correct behaviour based on implicit general knowledge. An example of such knowledge is that memory access errors are almost always considered incorrect behavior. [Barr et al., 2015]

The implemented test oracle is a combination of explicit, derived, and implicit test

oracle. The market messaging functionality is specified in GENERIS' functional descriptions and those specifications are used as a base for the oracle. The oracle also shares traits from implicit test oracles, since it is obvious that the time series values and statuses must be preserved correctly during the process. In the implemented test case, the oracle can also make the verdict for test failure without receiving any output. The test case has multiple states where it waits for some reaction from the SUT. If nothing happens before a time out limit is reached, the test is considered failed. The execution times for the tasks in GENERIS depend on various properties: the computational power of the server, the condition of the database and the amount of data to process. This means that the time out limits vary based on the same properties and it should be determined case by case. Thus, it can be argued that a part of the oracle is derived from SUT properties.

The research concerning runtime verification is closely related to this thesis. Zhao and Rammig [2009] present a model-based runtime verification framework that extends state-of-the-art runtime verification methods. The extended framework monitors the system and checks the consistency against the system model. Simultaneously, the system model is checked against the system specification. [Zhao and Rammig, 2009] Even though this is quite far ahead of the current state of the QM, it sets an example and a possible direction for future development. In order to develop QM towards in this direction the state models should be defined in more detail that was done in this thesis.

Black-box test automation is no new field of study. For example, Edwards [2001] suggest a automate black-box testing framework for system components. Their framework has a higher degree of automation than the QM framework, as their strategy is based on combining automatic generation of test drivers, automatic generation of test cases and automatic or semi-automatic generation of oracles. [Edwards, 2001] However, their framework is for testing of the components, while QM framework is for system testing.

Wissink and Amaro [2006] suggest that keyword-based test automation is the best solution for most environments. Multiple studies about test automation frameworks provide evidence that support the statement [Kim et al., 2009, Pajunen et al., 2011]. The findings of this thesis support the fact. The current state of QM framework doesn't have any support for a keyword-based approach. It was discovered during the thesis that designing and developing the test with purely C# code proved time consuming. Of course, the test developer's coding capabilities have a significant impact on the time consumption. Nevertheless, the time spent coding is time away from test designing and test design can be even more crucial than the actual test implementation, as is the case in the case example bug described in section 5.3.2.

Cervantes [2009] explores the use of test automation framework. The framework evaluation team considered three options: developing a framework in-house, using a consulting company to recommend a commercial framework and searching on the web for a commercial product. While this thesis focuses on an in-house developed framework, Cervantes explores an open source framework called Software Test

Automation Framework (STAF). It should be considered if STAF or some other third party framework would also suit Enoro's needs and be more cost-effective than developing and maintaining an in-house framework. Cervantes also compared developing the test with the framework to developing the test from the scratch using Python programming and concluded that using a ready test automation framework is beneficial.

4 Research material and methods

This section discusses the practical part of this thesis. The research has three distinct objectives:

1. Analysis of most critical processes to be verified
2. Feasibility analysis of the QM
3. Analysis of the effects of the QM on SQA process

4.1 Research Objective 1: Analysis of Most Critical Processes

In order to develop a useful software verification system the critical functionalities must first be identified. Since many operations of a DSO are determined by laws and regulations the main material for this research objective is the Finnish law. Additional information can be found in other government decrees and recommendations by Energiateollisuus. The analysis is based on the following materials:

- Sähkömarkkinalaki 588/2013 [Työ- ja Elinkeinoministeriö, 2013]
- Työ- ja elinkeinoministeriön asetus sähköntoimituksen selvitykseen liittyvästä tiedonvaihdosta (809/2008) [Työ- ja Elinkeinoministeriö, 2008]
- Tuntimittauksen periaatteita [Rissanen et al., 2010]

The laws, regulations and principles of hourly metering are analyzed qualitatively for the responsibilities of a DSO. After identifying the responsibilities, the relevant processes in GENERIS are presented formally as BPMN diagrams. The processes of GENERIS are derived from personal expertise and functional descriptions. So far, there hasn't been any formal analysis on the core processes of GENERIS so this analysis provides fundamental information that helps to prioritize the development of QM tests. The BPMN diagrams can also clarify the underlying processes which is beneficial for designing the tests.

4.2 Research Objective 2: Feasibility Analysis of Quality Manager

The second research objective is a feasibility analysis on QM framework. The research method for this objective is to demonstrate the feasibility by implementing a QM test case for one of the identified core processes, market messaging.

The main research material for this part is the Quality Manager framework. This section describes the QM framework. The framework itself was not developed as part

of this thesis so the framework is not analyzed in detail. Instead, it is presented on a high level in order to provide preliminary knowledge about the framework.

4.2.1 Quality Manager Framework

Quality Manager is a new software verification framework developed at Enoro. The core idea of the Quality Manager is that it can simulate the external electricity markets for GENERIS MDM system. This way a normal production behaviour of the critical processes can be verified and evidence of meeting the quality requirements is gathered during an execution run of the system. In other words, the QM test cases operate on the system level and they are purely black-box tests.

The QM architecture is described in figure 6. The test executor, human or automation, operates the QM task server system. Task server hosts multiple different QM test cases, or tasks as the name implies. The QM test case contains all the variables, methods and test steps that it utilizes during the execution. The methods and steps might also contain local variables.

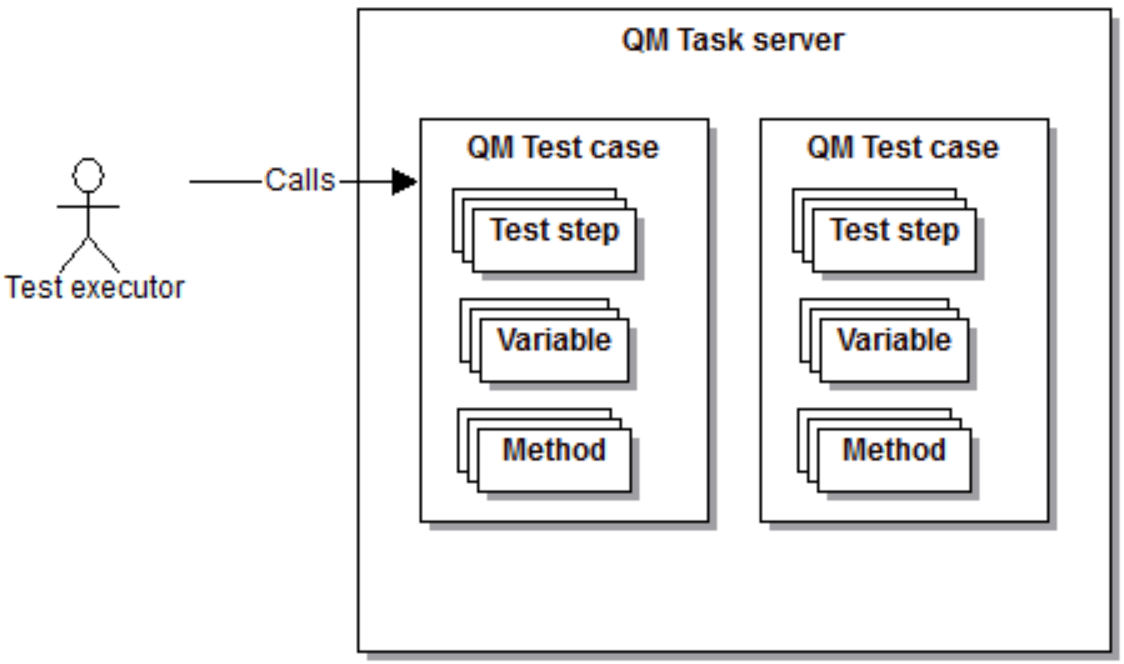


Figure 6: Architechture of Quality Manager

The QM framework features a time shifting functionality. This is useful, because the processes of an MDM system are often time dependent. Many of the processes feature timers and are executed once per day. One of the objectives of this feasibility analysis is to demonstrate the feasibility of the time shifting feature in testing.

The QM framework itself sets few requirements for the test. The test cases are developed in C# programming language and can be implemented in countless of

ways. It is up to the test developer to plan and implement a test design that supports the testing most effectively. The approach selected for the purpose of this thesis is a combination of runtime verification and black-box system testing. It is further described in section 5.2.1.

The test cases must implement certain interfaces so that QM task server can get information about the test during the execution. On the other hand, the test can interact with the environment using the same interfaces. The QM framework doesn't set any requirements for the test case except these interfaces. The test cases and the different features that are developed for the test case should be implemented as classes since C# is an object oriented programming language.

The interfaces that the test should implement are fairly straightforward. The interface *ITestCase* dictates that the test case class must have variables id, name, purpose, steps, inputs and outputs defined. First three of the variables are simply string type. The steps should be the number of test steps in the test case. Inputs and outputs are variables that contain the possible parameters that can be given to the test case when it is executed. Additionally, the test case should implement an Execute method that contains the actual execution of the test. The test can consist of one or more test steps that implement an interface class *ITestStep*. Additionally, there are interfaces *ITestContext* and *ITestEnvironment* that are used to access the execution parameters and the time related functionalities.

4.2.2 Features for AutoTester Comparison

After implementing a verification scheme in QM a comparison to the existing test execution framework, AutoTester, was conducted. AutoTester is a test automation tool developed at Enoro. It is meant solely for testing GENERIS. The following features were selected for comparison:

1. Lines of code
2. Development time
3. Test approach
4. Other differences

The material for the comparison is the developed QM test case and a similar test case developed in AutoTester. However, AutoTester doesn't have the time shifting functionality so the test is inherently different. On the other hand, QM doesn't have a way to manually execute IFM jobs from GENERIS because of its black-box nature whereas AutoTester operates the IFM jobs of GENERIS directly. For these reasons the two tests are not entirely comparable.

Even though the approaches aren't strictly comparable the general properties of the tests can be compared. The approaches utilize different testing methods and those differences can be analyzed. The amount of code lines reflects the complexity of

the implementation which is a comparable feature. Development time is somewhat comparable property since there was no previous experience of neither QM nor AutoTester. A lot of time was consumed for learning in both cases so the total development time doesn't reflect the actual time a dedicated test developer would use for the same task. In addition, there are typically more than one test developer creating the tests. In case of seasoned team of test developers, the time consumption comparison might yield significantly different results. Such a scenario could be studied in future research.

In this thesis all the development was done by a single person. If similar research is conducted when there ar

4.3 Research Objective 3: Effects on Software Quality Assurance Process

The third objective of this thesis is to analyze the effects that the QM framework has on the SQA process. The IEEE 730 standard is the main material used for the analysis. The goal is to analyze which activities of the standard SQA process are affected by the Quality Manager and how. Naturally, the concrete effects on quality, such as the amount of error findings after delivery and cost-effects, are long-term effects and can't be shown during the time frame of this thesis.

In addition, a recent error report is analyzed as a case example. The purpose of this analysis is to provide evidence for the implemented test case's value. The material is the actual error report in Enoro's error database. The printed version of the report is in appendix A. The error report was made anonymous by replacing all the identifying information with generic character sequences. The report is analyzed and the error is simulated for the test case by intercepting the normal test flow and making the necessary changes to the output of the SUT. The test is then expected to find the incorrect behavior.

5 Results

5.1 Critical processes of GENERIS

This section discusses the results for the first research objective that was to identify the critical processes of GENERIS based on the responsibilities of a DSO mentioned in the laws and regulations. Not all of these responsibilities are related to GENERIS. The identified responsibilities and the relevance to GENERIS are listed in table 1.

Table 1: Distribution system operator's responsibilities

Responsibility	**GENERIS relevant**
development of the grid (588/2013, sec. 19 §)	no
connecting metering points and production units (588/2013 20 §)	partly
distribution of electricity (588/2013 21 §)	no
arranging metering and registering the measurements for billing and balance settlement (588/2013 22 §)	yes
acquiring network loss energy (588/2013 23 §)	partly
publishing terms and pricing of services (588/2013 27 §)	no
planning for disturbances and emergencies (588/2013 28 §)	no
balance settlement (588/2013 74 §)	yes
notification responsibility (588/2013 75 §)	yes
balance settlement calculation and related information exchange (217/2016 section 4, 3 §)	yes
converting hourly readings to hourly powers [Rissanen et al., 2010, ch. 6.1]	yes
estimation [Rissanen et al., 2010, ch. 7.7]	yes

The responsibility to develop the grid mostly concerns the physical grid so it is not relevant to GENERIS system. Connecting the metering points and production units to the grid has two sides: physically connecting them and adding them to the MDM system for measuring. Therefore this responsibility is stated as partly relevant. Distribution responsibility means that the DSO must sell their distribution services for a reasonable compensation to those who require them.

The next responsibility, arranging the metering and registering the measurements, is very closely related to an MDM system such as GENERIS. Registering the measurements is one of the main purposes of GENERIS. GENERIS stores the measurements in time series that are linked to metering points. The measurements can be imported to GENERIS system via different interfaces that were described in section 2.3.2. The actual metering can be outsourced and GENERIS doesn't set any

requirements for the metering process, as long as the data is acquired in a supported format.

Acquiring network loss energy is listed as only partly relevant. The loss energy means the difference between the produced and consumed energy. The losses are natural conversion and transmission losses. The DSOs must have a contract with a supplier for the loss energy. The loss energy is defined by balance calculations that are done in the GENERIS but the actual acquisition is not done in GENERIS. Additionally, a DSO can use GENERIS to calculate the network loss for internal analysis.

Balance settlement is the next GENERIS relevant responsibility in the list. Balance settlement is carried out by the DSOs and the imbalance settlement responsible party. The process is specified in [Työ- ja Elinkeinoministeriö, 2008]. Balance settlement is the calculation process that balances the consumption and production throughout the electricity distribution system in Finland. This is one of the most important processes in GENERIS and it is formalized in figure 9.

The notification responsibility means that the DSO is responsible to make the necessary notifications to all the relevant parties about the results of balance calculations and electricity trade. The notifications should be made according to the regulated notification methods. [Työ- ja Elinkeinoministeriö, 2013] The notification responsibility concerning balance settlement is further specified in [Työ- ja Elinkeinoministeriö, 2008]. The notification responsibility also concerns the electricity trade. These notifications for the electricity supplier must be made per metering point [Työ- ja Elinkeinoministeriö, 2013, 22 §]. Notification responsibility concerns both the balance settlement process and the relaying of the measurements to suppliers. These processes are described in figures 8 and 7. The DSO is also required to send information to relevant parties when a supplier of a metering point is changed. The last reading of the metering point must be sent to the current supplier and an estimate for the yearly consumption to the new supplier [Työ- ja Elinkeinoministeriö, 2008, 8 §, 10 §]. Depending on the configuration, these notifications can be sent from the customer information system or MDM system.

There are also responsibilities that are not mandated by the law. One such responsibility is converting the hourly readings to hourly powers. This calculation is done in the MDM system [Rissanen et al., 2010]. Another such responsibility is estimating the missing measurements. Metering can fail due to temporary disruptions in the data transfer connection from the smart meter to the metering system. Sometimes the smart meter can even be broken. In such cases, the DSO must estimate the consumption. The estimation must be based on cumulative readings whenever they are available [Rissanen et al., 2010].

With this, the responsibilities of a DSO have been identified. In the following subsection the GENERIS processes involved in these responsibilities are formalized using business process modelling notation.

5.1.1 Business Process Modelling Notation Diagrams

Figure 7 shows the GENERIS process for delivering the measurements to market parties. First the DSO receives the measurement data through some of the metering data interfaces. The data is then saved to the database. After this, as a timer driven process, the system gathers the data that needs to be sent to retailers. According to Energiateollisuus Ry [2013] only the new and changes values should be sent. However, only whole days should be sent so if only one hour is changed, the whole day should be sent anyways [Energiateollisuus Ry, 2013].

After sending the data, the system passively waits for an acknowledgement message from the recipient. One MSCONS message to a market party may contain measurements for several metering points. These can be individually accepted or rejected within one acknowledgement message. If the acknowledgement was positive, the process ends and is repeated next day. If the acknowledgement was negative, GENERIS shows that the message got a negative acknowledgement and the operator should inspect the reason for the negative response and determine the correct actions to fix the situation. If the acknowledgement message is not received within time limit, the system shows that as well.

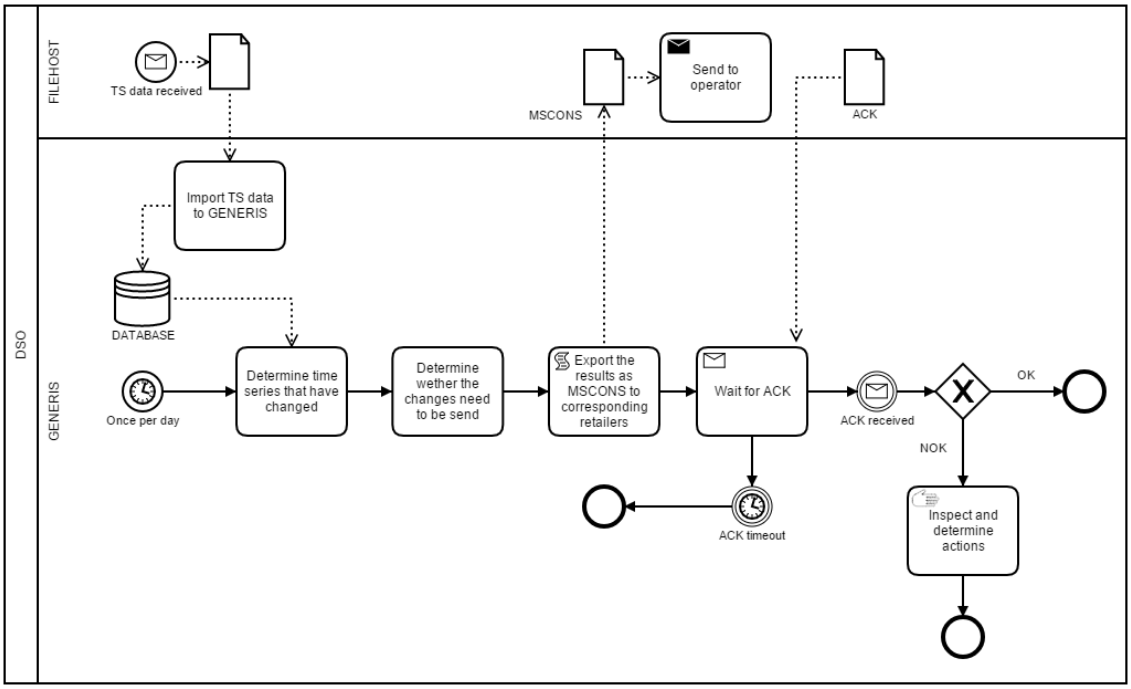


Figure 7: Typical message flow from DSO scope GENERIS to retailers

Information exchange concerning balance calculation was listed as one of DSO's responsibilities. The GENERIS process is similar to delivering the measurement data to retailers as can be seen from figure 8. However, while data to retailers is sent per metering point, the data that is sent to the imbalance settlement responsible party is aggregated per retailer.

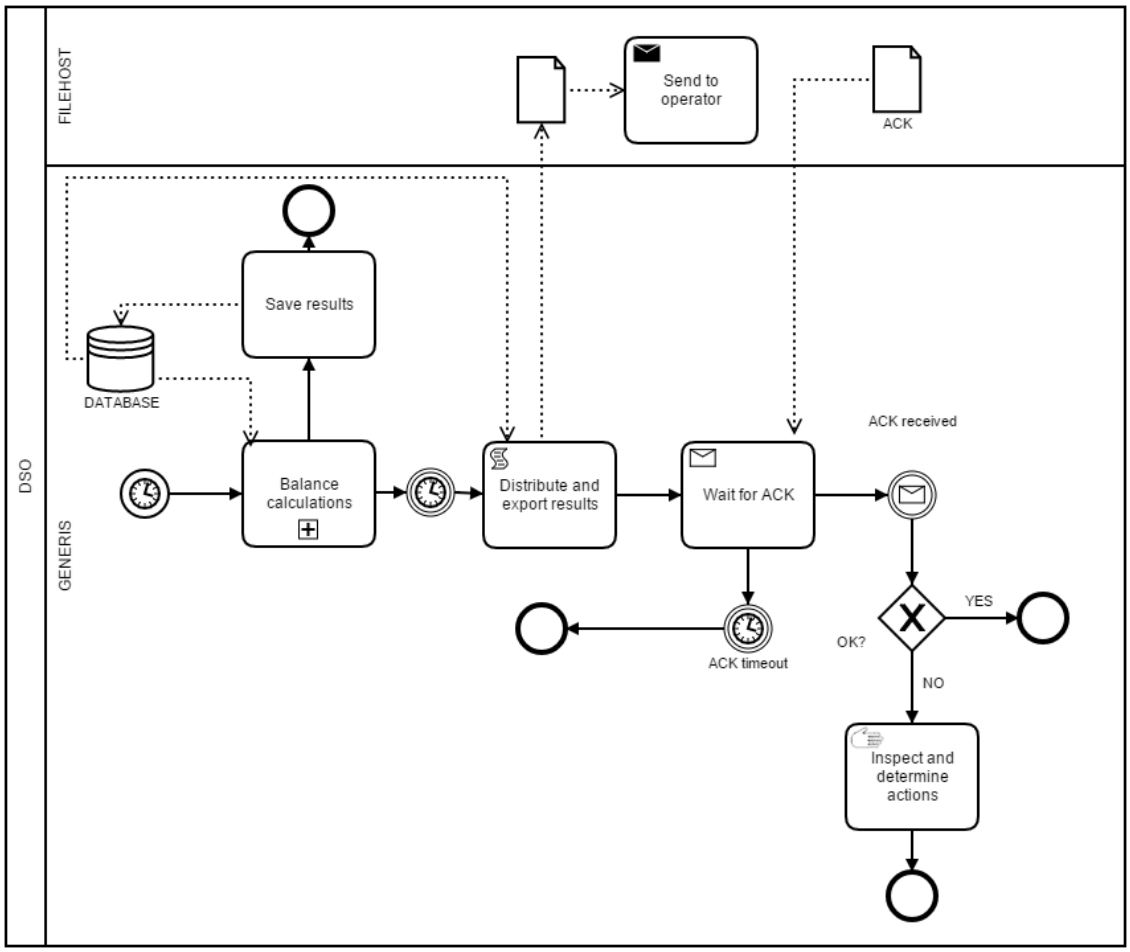


Figure 8: Message flow related to balance settlement

The process shown in figure 8 assumes that the metering point wise time series data is already acquired from the metering service and saved to the database, similarly as in 7. Once per day the DSO performs the balance calculations and saves the results to the database. After this, the results of the calculations are sent to the balance settlement responsible party.

The process in figure 8 features sub-process called balance calculations. These calculations mean aggregating the metering data for balance settlement and were mentioned as responsibility of a DSO. Therefore the calculations are very important process in GENERIS. Figure 9 shows the balance calculation process as it is in GENERIS.

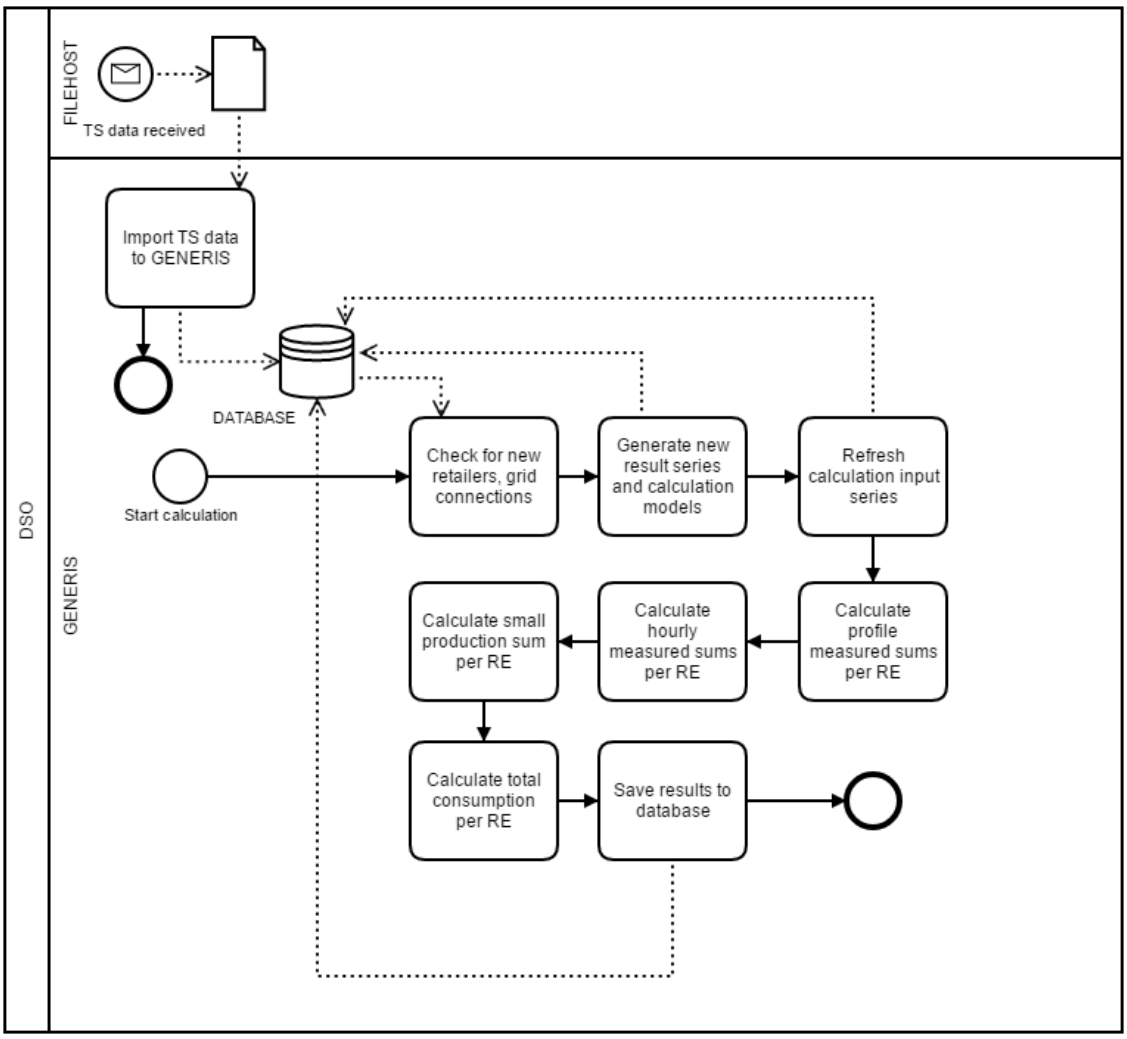


Figure 9: Balance calculation process in GENERIS

The GENERIS process concerning supplier change situations is highly dependent on the overall system configuration. Therefore, no single diagram can be drawn for it.

5.2 Feasibility of Quality Manager

This section describes the results for feasibility analysis for the QM. The analysis was conducted by implementing a test case for the market messaging process. The implementation is one of the core results for this research objective. First, the implementation is described in detail. After that the main differences between the existing test execution framework and the QM framework are presented.

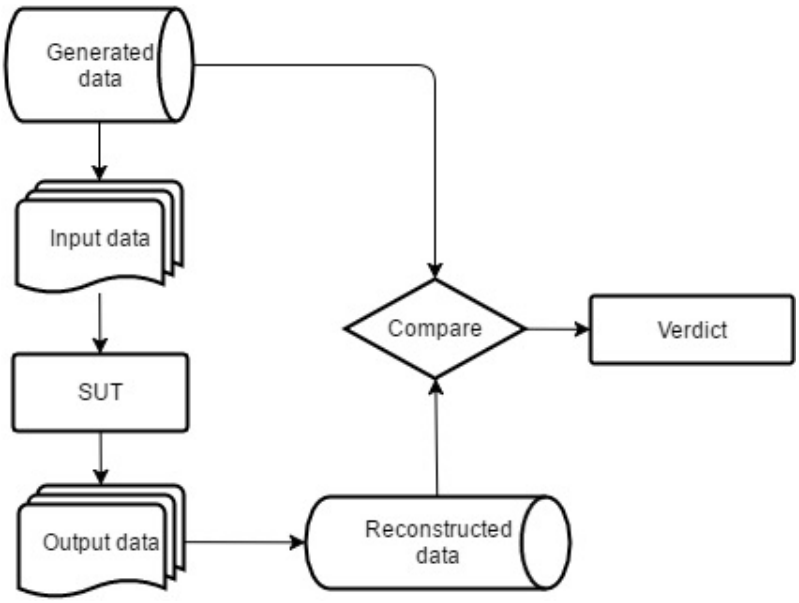
5.2.1 Test Implementation

As the Quality Manager features the possibility for time shifting it was decided that the market message testing should also cover a whole year. This was done in order to demonstrate the time shifting's feasibility in testing and to find possible problems with the developed functionality. The SUT is run for one year of virtual time and the test doesn't influence the SUT internally in any way. Therefore the test can be considered as runtime verification.

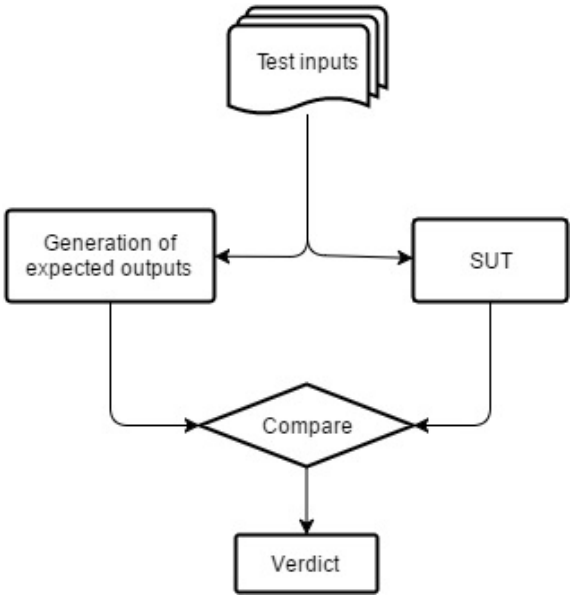
The approach is described in figure 10a. It is similar to the automatic test oracle approach but it doesn't feature generating the expected output. Instead, the output data from the SUT is reconstructed to QM format. The reconstructed data is then compared with the data that was used to generate the input. This comparison is then used for the verdict.

The figure 11 shows an UML class diagram describing the implemented test case. The highlighted classes *SAFToMSCONSYear* and *MSCONSRead* were developed within this thesis. The figure shows that the test case *SAFToMSCONSYear* implements the interface ITestCase. The test case also has fields and methods that are used internally by only this test case. The test case has variables for the ACC and SAF input folders and the MSCONS output folder. Time out limits for SAF import and MSCONS export can also be given as input variables. Those are given for the test case as command line parameters in the Quality Manager command line user interface.

The test case consists of multiple test steps for every day that are stored in the list variable TestSteps. The variable inputData holds the randomly generated time series data that is used in the test.



(a) Implemented test approach



(b) General automatic test oracle

Figure 10: Implementation compared to general automatic test oracle

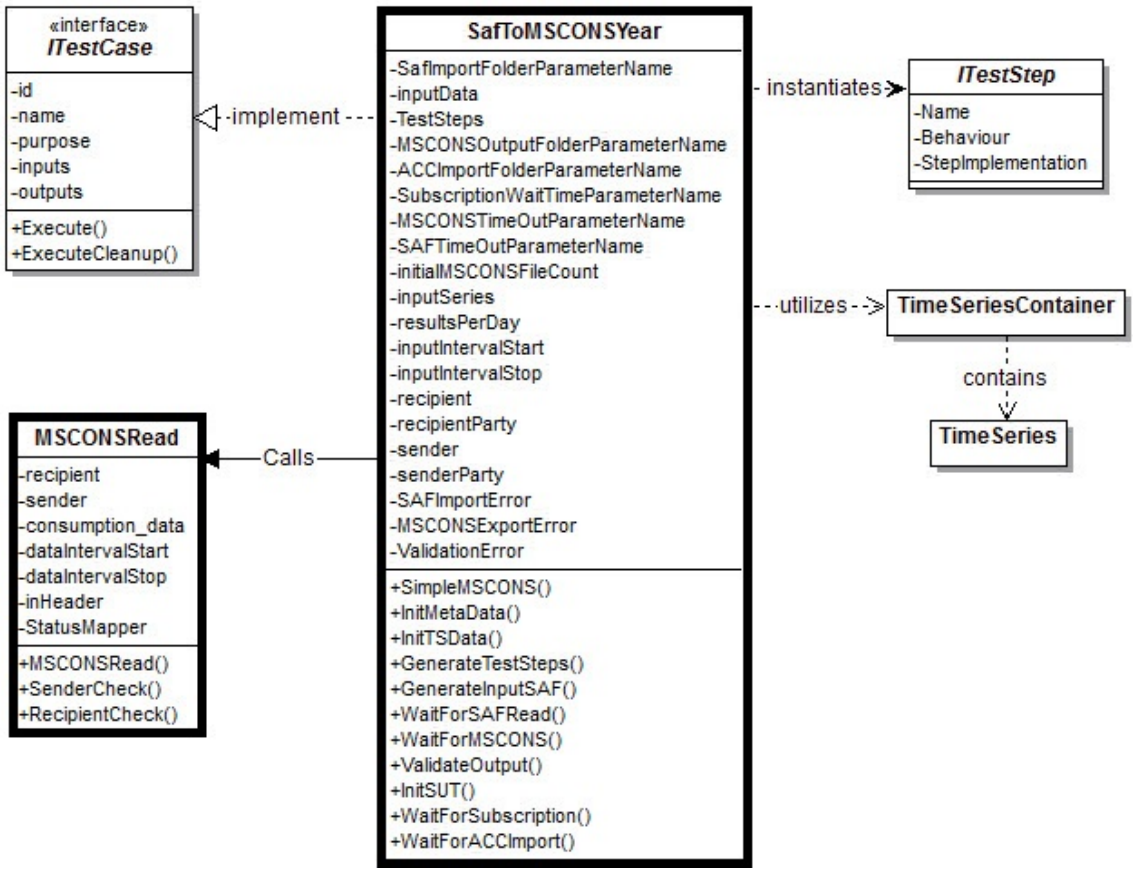


Figure 11: UML class diagram of the implemented QM test case

Multiple methods were developed for the test case with various purposes. The method *InitMetaData* was used to make the test constructor cleaner. It initializes the fields that are required by the ITestCase interface. The *InitTSData* method randomly generates the time series data for one year that is the time period used in the test case. *GenerateTestSteps* method generates multiple test steps for each day of the test's time period. The methods *GenerateInputSAF*, *WaitForSAFRead*, *WaitForMSCONS* and *ValidateOutput* are implementations for the test steps that were generated with *GenerateTestSteps*. The remaining method *ValidateOutput* is called within *WaitForMSCONS* and is used to validate the output MSCONS file. These methods are utilized during the test execution. The figure 12 shows the general activities during the test execution.

The test case constructor is called when the Quality Manager task server is launched and is therefore not showed as a part of the activity diagram in figure 12. InitMetaData method is called within the constructor. Additionally, dummy input parameters are added to the inputs list. The input parameters don't really receive any values at this point. The actual parameter values are received from the task server after the test has been called.

The Execute method is called when the test is signaled from the Quality Manager task

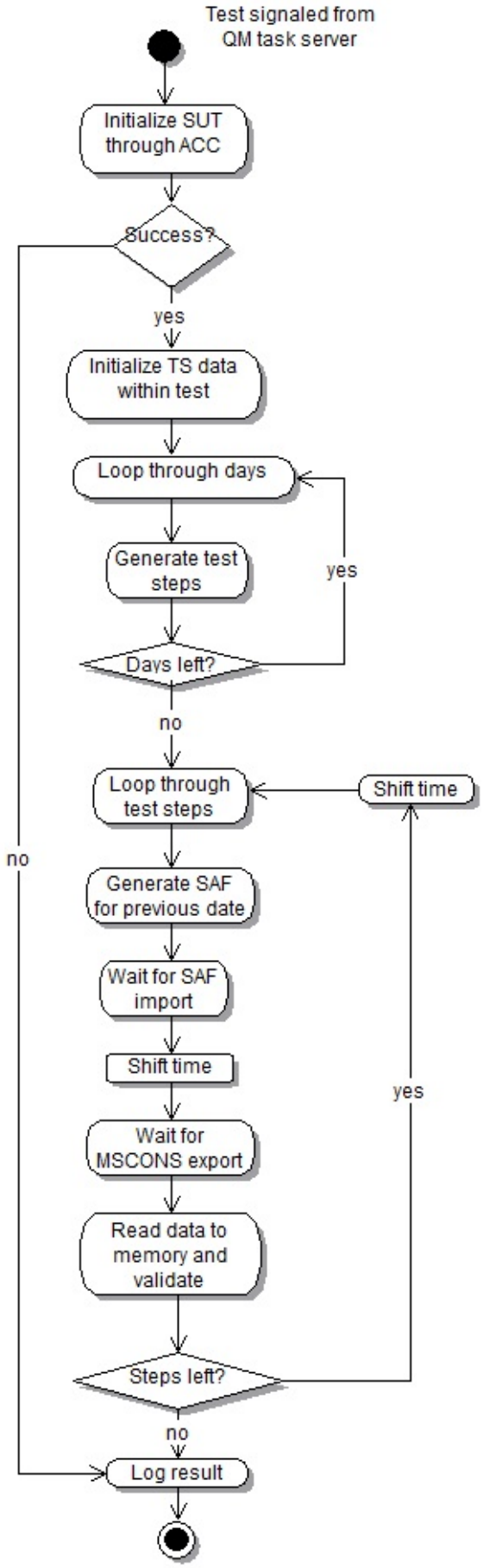


Figure 12: UML activity diagram for the test execution

server. First, the SUT is initialized using ACC interface and preconstructed ACC files. Initialization in this case means that the master data structure is generated in GENERIS database. This is a precondition for the test. Market messaging requires that there is a correctly configured metering point with an active measurement before any messages are sent to market parties. Figure 13 describes the data structure and preconditions that are initialized during the initialization step. Some of the properties, such as the address of the metering points, are optional but before the test can work the objects themselves must be initialized.

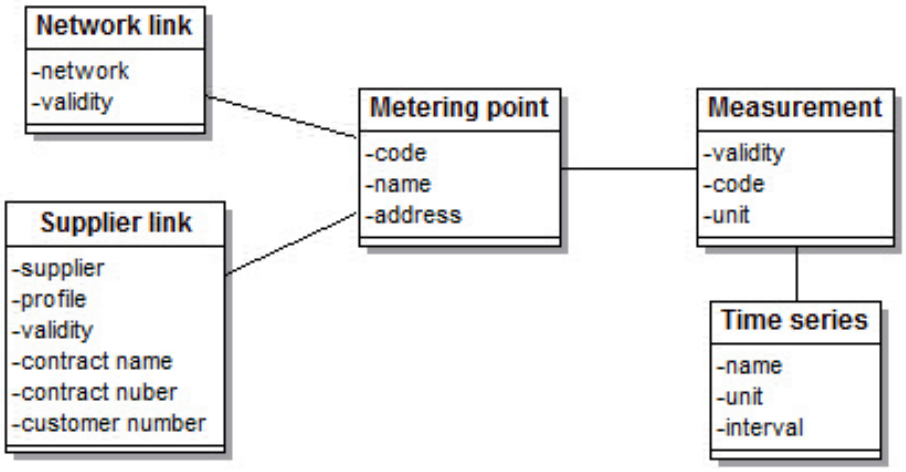


Figure 13: GENERIS metering point data structure

The test case is designed to test market messaging for a single hourly metered metering point. It is mandatory that the supplier link is properly initialized. The validity of the supplier must contain the test period and the consumption profile must be hourly metered. The customer information is optional but it is included in the ACC files nevertheless. The metering point must also have a valid interval measurement with interval time series linked to it. Network link must be set to the network of the DSO of the SUT.

Time shifting is also utilized in the initialization phase. In order to detect the changes in time series GENERIS needs to add so called reprocessing subscriptions to the time series. In short, the subscription is the entity that keeps track on which periods of the time series have changed after the last message containing said time series was sent. The subscriptions are added in a process that is started with a time, so after creating the master data the time is shifted to the next time the subscription creation process would launch.

The next step is generating the time series data. The method InitTSData initializes the time series data for one year and stored it within an internal variable. The data is generated and stored within the test and this activity doesn't affect the SUT in any way. In this test case, the consumption data is random values between 0.1 and 5. All of the statuses for the measurements are set to 'measured' status.

The next activity is generating the test steps. Three steps are generated for each day of the test:

1. Generate input SAF file
2. Wait for SAF import
3. Wait for MSCONS export

After generating the steps, the execution loops through all the steps in order. If there is a special time behaviour attached to the test case, the time behaviour is passed on to the test environment and the environment shifts the virtual time accordingly. First test step for each day is generating the input SAF file. This step always shifts the virtual time to the next date before executing the step implementation. The SAF file is generated based on the time series data stored within the test. The consumption data of the previous virtual date is used to generate the file. The file is created into the GENERIS's SAF import folder.

Next test step is just to wait for SAF file to be imported to GENERIS. GENERIS moves the file after importing, so the test knows that the file has been imported if it is no longer present in the import folder. If the file doesn't disappear within the specified time limit, the test step time outs and the test case aborts and is marked as failed.

After waiting for SAF import the test waits for MSCONS export. The time is shifted to the time of market message distribution that is configured in the SUT. The distributions are done once per day at a certain time so the time shift is required to make the test feasible. A time out for this step is also given as an input parameter for the test. If an output MSCONS file is found, it is read and validated at the end of this step.

An additional class *MSCONSRead* was developed for validating purposes. It reads and stores the data from an MSCONS message into internal variables. The whole file is read into a string array and those strings are looped through. Based on features present in the strings the script determines whether information should be stored. If there is data to be stored, the relevant data is extracted from the string. The MSCONS messages are standardized so the strings or segments always have the information in the same place.

After utilizing the MSCONSRead to extract data from the outgoing MSCONS it is validated against the data that was used to generate the input SAF file. The validation workflow is presented in figure 14. The following aspects are considered:

1. Recipient
2. Sender
3. Data interval
4. Time series name

5. Time series values
6. Time series statuses

The sender and the recipient are simple to check from, for example, the UNB segment of an MSCONS message [Ediel, 2002]. They need to match the DSO of the SUT and the Ediel address of the test metering point's supplier. The time series name is written in the LOC segment of an MSCONS message and can be easily verified [Fingrid, 2015]. The time series name in the MSCONS message also contains the network and supplier codes [Fingrid, 2015]. Therefore, it can be used to check that the message is consistent with itself.

In this test case, the data interval should be the whole previous virtual day. In more complex tests it can be something different. In such a case, the interval must be determined from the input data according to specification. The output data interval can be determined from the DTM segments in the header part of an MSCONS message or naturally from the time series values themselves [Ediel, 2005]. Once again, these two different ways to check the data interval can be used to verify the consistency of the message.

Checking the time series values and statuses is fairly straightforward. The values and statuses are stored in a time series data structure within class MSCONSRead. They are compared time stamp by time stamp with the value and status that was used to generate the original SAF file.

If all validations are passed, the day is considered a success. If even one validation failed, the test is marked as failed and an error message is written to the test log. The message contains information about the failed validation and the virtual date of the failure. No failures are accepted so in case of validation failure the whole test case is aborted.

The implemented software verification approach is most accurately described as a combination of black-box system testing and runtime verification. The QM doesn't interfere with the execution of GENERIS in any way even if it detects an error in the execution which is typical for runtime verification [Leucker and Schallhart, 2009].

Consider the division of the monitors by Delgado et al. [2004] that was presented in section 2.2.3. In this implementation the monitoring points were placed manually. Placement was offline, as the all the code for testing was separate from the SUT's coding. Platform was naturally software, as no additional hardware was used in the verification. The final classifying feature, implementation, was multiprogramming in this case. The QM is was executed on the same server with the SUT, but as a different process. QM framework also allows the multiprocessor approach, but in such implementation the time shifting wouldn't be available.

On the other hand, the approach can be considered as software testing since the QM hosts strictly defined test cases that include one or more test steps. Since the testing is done with a fully integrated complete system using only normal input and output

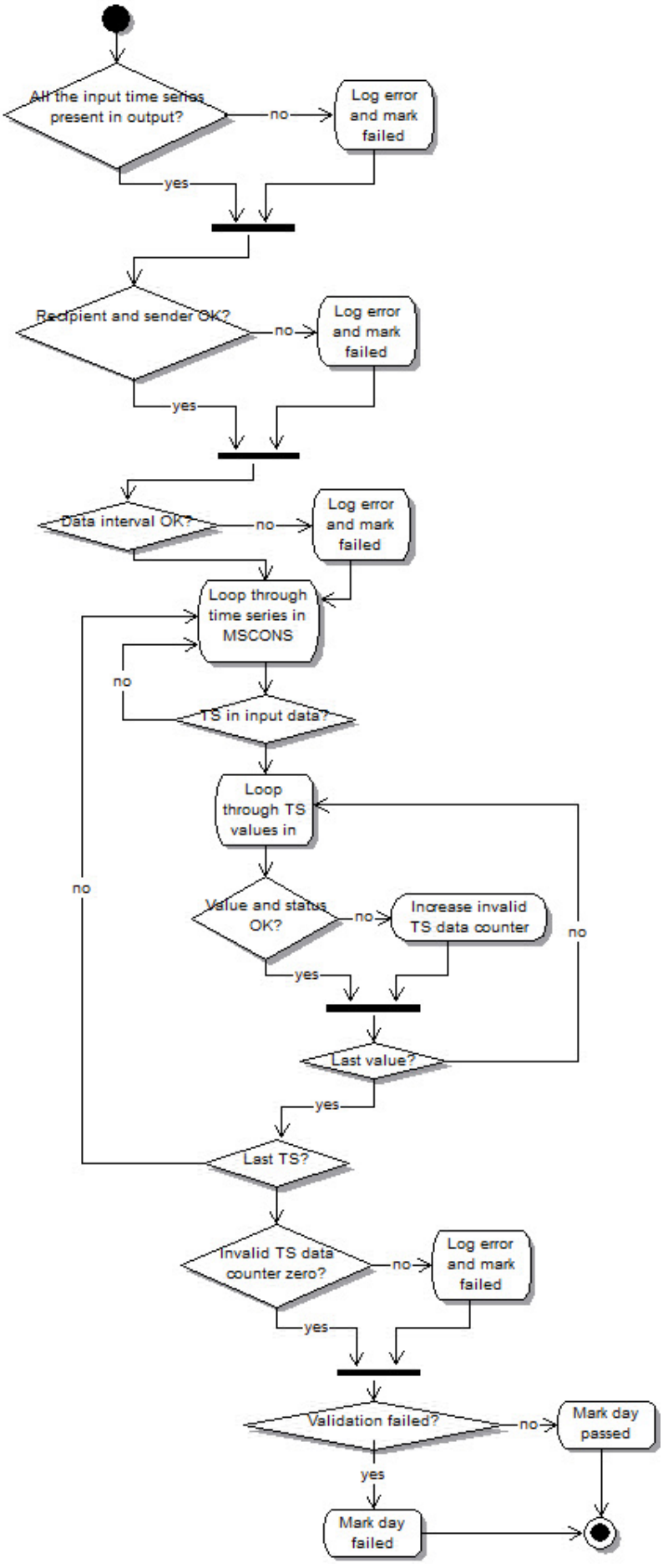


Figure 14: Validation workflow

interfaces the testing approach is black-box testing at system level.

The approach has traits from multiple software testing methods that were described in 2.2.1. The inputs that are given to the SUT are all similar positive random values. Therefore the test actually covers a single equivalence class. It would be possible to incorporate more classes within the same year but this is left for future development. On the other hand, the test case uses a form of state transition testing. The time outs imply that the SUT didn't go to a correct state. The time outs are used to make the verdict so the method has traits from state transition testing as well.

The executions of the implemented test case are discussed in the next section.

5.2.2 Formalizing test requirements

The monitors that are used in runtime verification are based on requirements, as was discussed in section 2.2.3. The implementation for the test case that is described in section 5.2.1 is based on these requirements.

In fact, parts of the test implementation can be identified as monitors. Some of the monitors are based on temporal requirements, while others are implicit and based on non-temporal requirements. This section identifies the requirements for the monitors that are present in the test implementation. The requirements are first described in natural language and then formalized with using metric temporal logic for the temporal requirements and common propositional logic for the non-temporal requirements.

The first requirement is related to the system initialization step, where the master data is created with ACC files: *If an ACC file is created in the ACC import folder, the file must be moved to the done folder within time t.* In MTL, this can be written as:

$$\Box \left(\Diamond a \rightarrow \Diamond_{[0,t]} (\neg a \wedge d) \right) \tag{1}$$

where a = *ACC file present in import folder* and d= *ACC file present in done folder.* This requirement can be applied to all different ACC files that are used within the initialization phase of the test.

The temporal requirement for SAF import is equivalent to the previous requirement: *If a SAF file is created in the SAF import folder, the file must be moved to done folder within time t.* This can be formalized as:

$$\Box \left(\Diamond s \rightarrow \Diamond_{[0,t]} (\neg s \wedge d) \right) \tag{2}$$

where s = *SAF file present in import folder* and d = *SAF file present in done folder.*

The temporal requirement for MSCONS messaging is dependent on the SAF file import requirement. Consider the requirement: *If a SAF file was successfully read, an MSCONS message must appear in the output folder after time t_1 but before time t_2*. The implication is that the MSCONS message distribution must start after time t_1 and it must be finished before time t_2. In the implemented test, time shifting is used to skip straight to time t_1 without waiting for the actual time to reach it. The t_2 sets the performance requirement in the distribution. If we replace the requirement 2 with σ, the MSCONS requirement can be written as:

$$\Box\left(\Diamond\sigma \rightarrow \Diamond_{[t_1,t_2]}m\right) \tag{3}$$

where σ means that the SAF import that fulfilled requirement 2 has occured and m = *MSCONS-message exported to output folder*. Generally, this requirement could be fulfilled even if the time limit of the SAF import was violated. However, the test was implemented in a way that if even one of the requirements is violated the monitoring is stopped. Therefore, the fulfilment of the requirement 2 is a pre-condition for even monitoring requirement 3.

The rest of the requirements have no temporal features. They only address the features of the output file. The first requirement regarding the contents of the MSCONS message is: *If time series x was present in the input, time series x must be present in the output*. This can be formalized by basic group theory. If X is the set of input time series and Y is the set of output time series the requirement can be written in propositional logic as:

$$x \in X \rightarrow x \in Y \tag{4}$$

The second non-temporal requirement is related to the senders and recipients of the message. The requirement can be stated as: *If the supplier of the metering point was initialized to s, the recipient of the MSCONS message must be s.*

$$A = s \rightarrow B = s \tag{5}$$

where A is the supplier that was initialized with the ACC files and B is the recipient of the MSCONS message.

In comparison to the requirements 1-4, the requirement 4 is specific for this test. In true production use there would be supplier changes occurring due to residents making new contracts for electricity supply. In such cases, the recipient of the message would be related to the validity of the supplier link of the metering point. In some cases it can even be required to send the same measurements to two different recipients. However, such scenarios were not considered in this thesis as they would make the test more eager and are therefore left for future test development.

The requirement for the data interval is similarly specific for the implemented test case only and cannot be generalized universally. The requirement is: *If input data interval*

was $[t_1, t_2]$*, the output data interval must be* $[t_1, t_2]$. Let T_{input} be the data interval of the input and T_{output} similarly the data interval of the output. The requirement is then:

$$T_{input} = [t_1, t_2] \rightarrow T_{output} = [t_1, t_2] \tag{6}$$

In real production use the data interval requirement is not as straightforward. The output data interval depends also on the supplier link. In addition, [Energiateollisuus Ry, 2013] mentions that the whole days of data should be sent even if the measurements are changed only for single hours. In this case, the input data interval is not the same as the output data interval.

It was decided to demonstrate the feasibility with this simple requirement, which holds in a special case where supplier is constant throughout the whole year and only full days are present in the input data. The test can be generalized in the future but this would make it more eager. Therefore, making specific tests for different data interval scenarios should be considered.

The requirement for the actual time series data, however, is general. The implicit requirement is: *If input data with time stamp t is x with status s, the output data with time stamp t must be x with status s.* Formally:

$$I = (t, x, s) \rightarrow O = (t, x, s) \tag{7}$$

where I is the input time series data and O is the output time series data.

These are the formal requirements for the monitored features in the implemented test. The actual monitors are parts of the implemented test. The temporal requirements are typically monitored by the part of the test that wait for specific files while the non-temporal requirements are verified within the *ValidateOutput* method.

5.2.3 Test Execution

The execution of the test case was successful. The correct behaviour was indeed detected as correct behaviour. The incorrect behaviour was also correctly detected, as is described later in sections 5.3.2 and 5.3.3. Executing a test is intuitive. A command to start a certain task is given from QM task server command line interface. The parameters for the test are given after the command. Figure 15 shows a picture of the interface right before the execution.

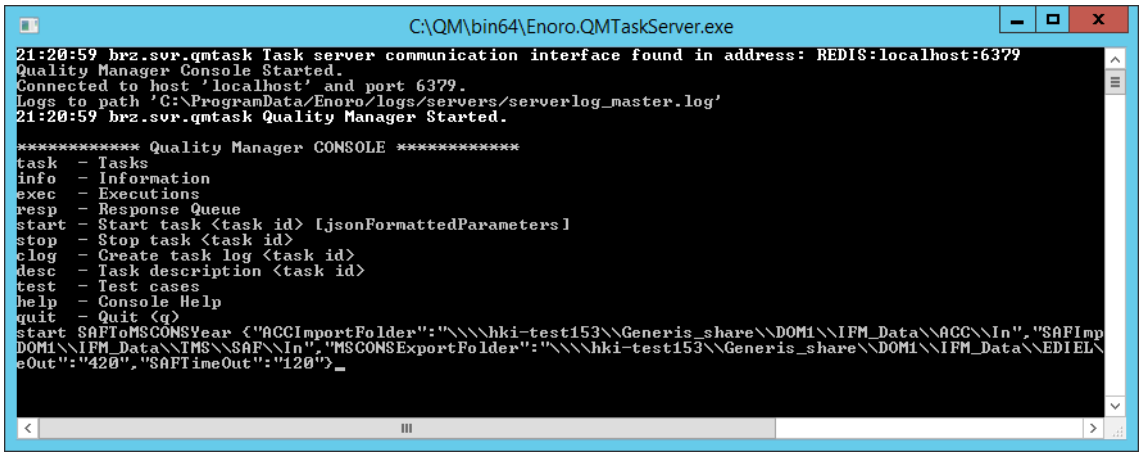


Figure 15: QM Task Server before test execution

The test executions were able to detect an error in GENERIS IFM implementation that made time shifting unusable in testing. The first one-year test execution ran fine for 23 virtual days. On the 24th virtual day, the generation of the MSCONS message took longer than expected. The time out limit for MSCONS generation in the test was set to three minutes at that point. Three minutes should have been enough for exporting one day of data for one time series. Similar behaviour was detected with longer time out limits and on random dates. Time outs happened during different test steps as well.

It was discovered that there was a communication problem with an IFM runner. The communication problem emerged when time shifting was done at a wrong time and the service pinging the IFM runner saw too long a duration between current time and last successful ping. This was due to shifting the current time one day ahead. The error was fixed and the time shifting functionality seemed to be working correctly after that. Indeed, the scheduled IFM jobs in GENERIS are executed automatically during the test execution. The execution logs of GENERIS also match the virtual time that is set within the test.

However, the test case did not succeed in running for the whole year. The test ran successfully from 28th October 2016 to 26th March 2017 (virtual time). On 26th March 2017, which is the date when the daylight saving time begins in 2017, the outgoing MSCONS was not exported within time limit. This appears to be an error in IFM job timer system. The error is further discussed in section 5.3.3.

Most of the time during a test execution seems to be consumed by waiting for IFM jobs. This can be seen from figure 16 that shows an ongoing test execution. The three parts that consume most of the time are waiting for subscription invalidations, waiting for SAF import and waiting for MSCONS. This leads to a conclusion that modifying the test case won't make the execution faster.

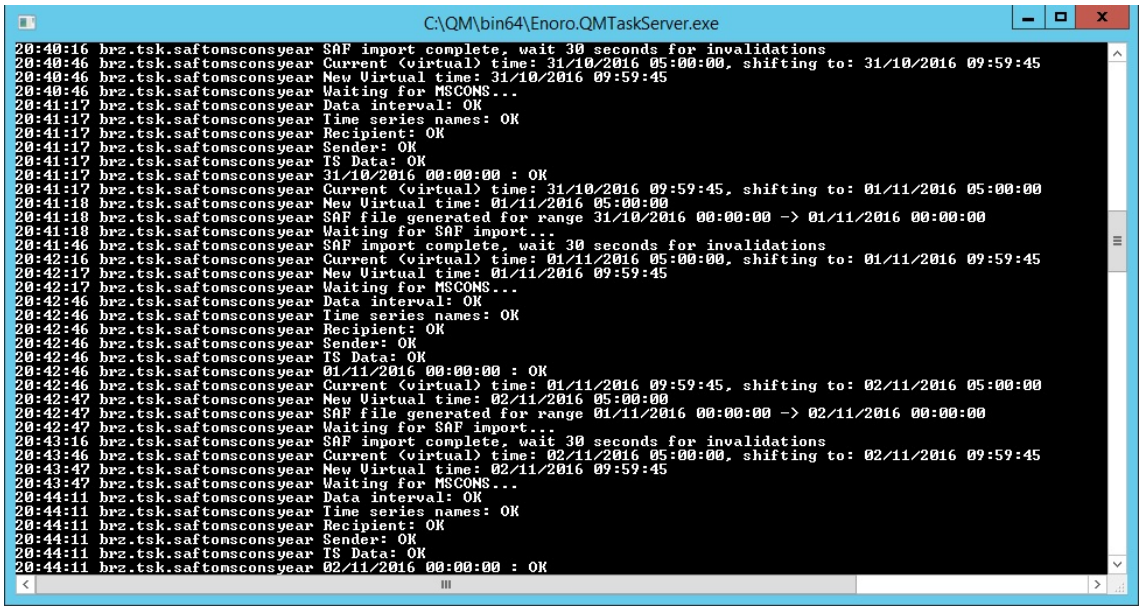


Figure 16: Ongoing test execution

5.2.4 AutoTester Comparison

This section discusses the differences between the existing test execution platform and the new QM framework. The approaches are quite different so this comparison is not enough to prove either better or worse.

Test cases for AutoTester are developed using a scripting language that is specific for AutoTester. An AutoTester test case to verify the market messaging process was developed for comparison. AutoTester doesn't have any internal variables, so it cannot be used for input data generation. Therefore all the input data and the reference output data must be preconstructed.

AutoTester has access to GENERIS' internal functionalities. It could be used to create the metering point without using the ACC interface. However, in order to make the tests as comparable as possible it was decided that the SUT should be initialized through ACC. The AutoTester script first copies the three ACC files to GENERIS' ACC import directory. There is one minute time out for each file. After importing all the files, the script signals an IFM-job to create the reprocessing subscription so that the freshly created metering point would be included in the market messaging.

After the SUT initialization phase is done the script copies a preconstructed SAF file to GENERIS' SAF import directory and waits for the file to get imported. Then the script waits for 10 seconds so that the subscription notices the change in the time series data. After the wait, an IFM-job for distributing the MSCONS messages is signalled.

The script then polls the MSCONS output folder for a message to the right recipient.

The message is then compared with the preconstructed reference message.

The AutoTester test implementation is thus very similar to the QM implementation. The main difference is the lack of input data generation and time shifting. Another significant difference the fact that the relevant IFM jobs are signalled from the script. From GENERIS' point of view this represents manual execution since the IFM jobs are normally executed automatically with timers.

Table 2: AutoTester and Quality Manager comparison

	AutoTester	**Quality Manager**
Lines of code	67	666
Time investment	3 h	50 h
Automatic input data generation	no	yes
Testing approach	Test execution automation	production simulation
Script parametrization	const strings in the script	input parameters

Table 2 shows the compared features. It can be seen that AutoTester scripts are far less complex than QM scripts. In addition, the time investment is a lot smaller. However, there is a difference in script maintenance.

If an AutoTester script is executed on a different SUT than usually, the whole script must be modified and verified again. The script may contain constant string variables but they must be replaced within the script. This increases the script maintenance required to keep the tests active. Some parts of the script may remain the same but the test must be reviewed regardless. QM test cases can be developed in a way that they are executable on virtually any instance of the SUT. The input parameters of a QM test can be used to take the variability between the systems into account. Therefore, AutoTester is most applicable to constant internal test servers, while QM can be used to verify the customer test systems as well. Of course, this requires a good test design with proper parameters. The QM tests can also be hard coded to work on a single server but this would be a waste of potential.

The lack of input data generation also generates more maintenance work for AutoTester tests. In this particular case, all the preconstructed ACC- and SAF-files, as well as reference MSCONS message must be modified if the test is executed on different dates. The time affects how GENERIS is supposed to work. ACC files initialize the supplier link to start from a specific date and GENERIS sends all the consumption data from the supplier link start date up to the current date to the new supplier. Naturally, this affects the output MSCONS message and the test will fail because of the static reference MSCONS message. The QM test can mitigate the different execution dates by generating the input data accordingly.

QM tests are more similar to actual production use. All the mentioned processes in GENERIS are automatic. This means that the automation of the processes

is a core part of the correct functionality. Since the AutoTester scripts represent the manual execution of the processes the automation requirements are completely neglected.

5.3 Effects on Software Quality Assurance Process

This section discusses the effects that the implementation of Quality Manager has on the SQA process. First the effects on general SQA process that was described in section 2.1 are discussed. After this, two case examples showing how the implemented test improves the quality are presented.

5.3.1 Analysis on Affected Software Quality Assurance Activities

The Quality Manager has a profound impact on the software quality assurance process as it operates on multiple levels of the SQA process.

Most important activities affected by the QM are the product assurance activities. Evaluating the product for conformance to the requirements is arguably the most important sub-activity of product assurance. The QM has a great impact on the methods that are used in this activity. The current test automation system focuses on executing certain tests by directly operating the SUT.

The QM approach is quite different. The QM approach is to identify the underlying business process of the DSO and utilize the relevant interfaces to simulate actual production behaviour. The total number of discovered problems is one of the most important drivers for customer satisfaction [Buckley and Chillarege, 1995]. Therefore, ensuring the quality of the actual processes that the customer observes will increase the customer perceived quality and customer satisfaction.

However, the software requirements can be very detailed and defined for smaller parts of the actual process. In this sense observing the process as a whole might left some requirements overlooked. Of course this depends on the implementation of the test and the transparency of the process. For example, a part of the market messaging process is importing the time series data from the metering service. A functional requirement might state that the time series values and statuses are saved to the database according to the received data file. The test that was implemented in this thesis has no way of actually verifying this requirement. It might be that the MSCONS export is faulty and it is a mere coincidence that the outcome matches the input. Of course the probability of such scenario is diminished by repeating the test for the whole year with random inputs.

The most natural application for the QM framework is evaluating the product for acceptance. This activity aims to provide supplier confidence that the product is acceptable to the acquirer. The acquirer is interested in the reliability of the actual

processes so the QM approach supports the acceptance testing activities extremely well.

The QM also affects the SQA process implementation activities. As a new tool is introduced to the SQA process the whole SQA process should be refined. The appropriate applications of the QM should be defined as part of SQA planning. The information provided by this thesis can be utilized in this activity. As part of the implementation process it is also important to coordinate with other processes. It should be considered whether the QM makes some tests redundant. Redundancies should be eliminated while considering the previously discussed fact that the QM can't be trusted for verifying all individual functional requirements. The functional requirements should be reviewed in order to determine which requirements can be sufficiently verified with the QM.

Naturally, the outcome of the refined SQA planning should be documented. The SQA organization needs to know what is tested and how. The testing for all new development must be planned with keeping the new methods in mind.

5.3.2 Case example: MSCONS Messaging Bug

During the pilot use of GENERIS branch 2.11.1 a critical bug in market messaging was discovered. The internal error report is attached in appendix A. Due to the bug, the sender and recipient of the MSCONS messages were mixed. The reason was that the new GENERIS version sorted the recipients in an XML file while the MSCONS export script was hard coded to pick recipient from a specific location in the XML.

The bug was fixed and it is no longer present in the version that was used in this thesis. Therefore, the error was simulated. The test case takes the MSCONS export folder as an input parameter when the test is executed. The test was given a different directory as a parameter than the actual MSCONS output folder of the SUT. This way it was possible to mix the sender and the recipient of the message before giving it to the test for validation.

The test case detected the error as expected. The detection is shown in figure 17. Of course, this kind of error could have been detected with the existing test automation system as well. It is a matter of test design. In this case, the QM framework itself doesn't add any additional value. However, since such a test was not previously implemented, the developed test case clearly is valuable.

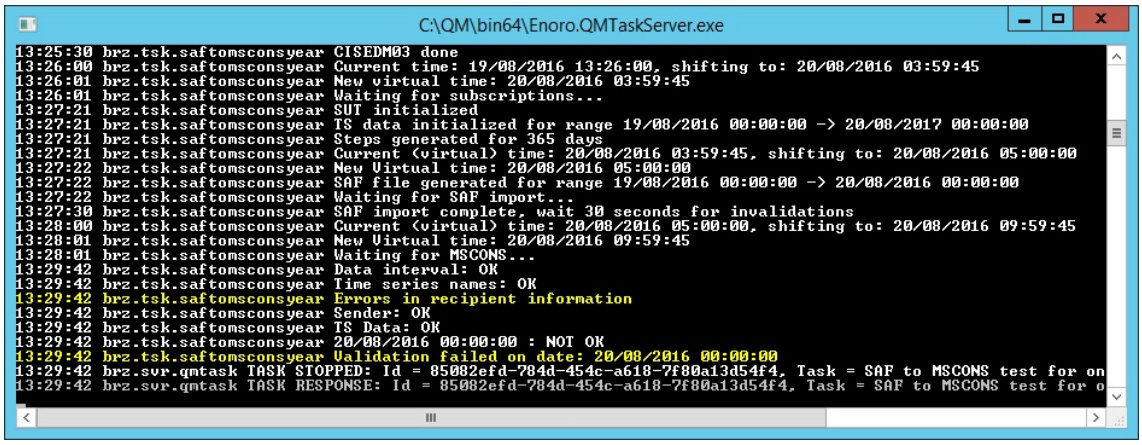


Figure 17: Detection of a wrong recipient

The previous test assumed that the MSCONS file name is preserved as expected. By default, GENERIS also writes the name of the recipient to the outgoing MSCONS message's file name. The test case detected a problem in this case as well but in a different way. The test case expects a message directed at a certain recipient. If such a file is not exported within the a specific time limit, the test fails. In this case, it remains for a human to inspect the cause of the test failure.

5.3.3 Case example: IFM job timer error in daylight saving time shift

An error with IFM job timers was discovered during the test executions. On 26th March 2017 the MSCONS message was not exported within the time limit and the test raised an error about it as can be seen in figure 18.

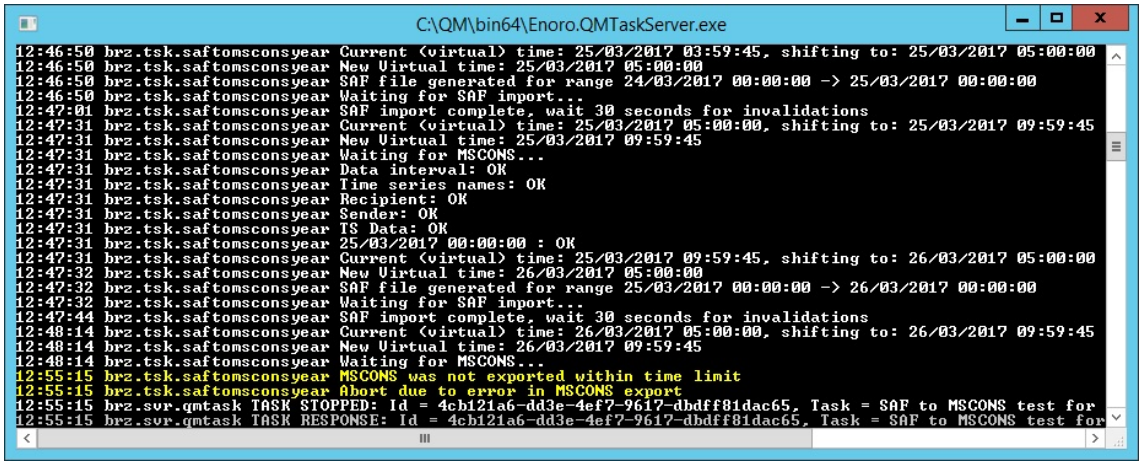


Figure 18: MSCONS export time out on 26th March 2017

The execution ran successfully before this date so it could be determined that there is a problem with the daylight saving time change. The MSCONS export process

was scheduled to execute at 10:00 EET+ (summer time) but it was never started. However, later inspection on the history journal showed that the job was in fact started at 11:00 EET+. This can be seen from figure 19.

	>0	Kaikki	-		
TMS2:_Sanomavälityksen_muuto...	38	Onnistunut	26.03.2017 11:00:01 - 26.03.2017 11:01:01	1m 0s	Olemassa...
TMS2:_Sanomavälityksen_muuto...	34	Onnistunut	25.03.2017 10:00:01 - 25.03.2017 10:00:51	50s	Olemassa...
TMS2:_Sanomavälityksen_muuto...	30	Onnistunut	24.03.2017 10:00:01 - 25.03.2017 05:00:04	19h 0m 3s	Olemassa...
TMS2:_Sanomavälityksen_muuto...	26	Onnistunut	23.03.2017 10:00:01 - 23.03.2017 10:00:27	26s	Olemassa...
TMS2:_Sanomavälityksen_muuto...	598	Onnistunut	21.03.2017 10:00:01 - 21.03.2017 10:00:35	34s	Olemassa...
TMS2:_Sanomavälityksen_muuto...	594	Onnistunut	20.03.2017 10:00:01 - 20.03.2017 10:00:25	24s	Olemassa...
TMS2:_Sanomavälityksen_muuto...	590	Onnistunut	19.03.2017 10:00:01 - 20.03.2017 05:00:02	19h 0m 1s	Olemassa...
TMS2:_Sanomavälityksen_muuto...	586	Onnistunut	18.03.2017 10:00:01 - 18.03.2017 10:00:37	36s	Olemassa...
TMS2:_Sanomavälityksen_muuto...	582	Onnistunut	17.03.2017 10:00:01 - 17.03.2017 10:01:15	1m 14s	Olemassa...

Figure 19: IFM history journal

This is a genuine error and requires further inspection from the developers. This thesis has therefore contributed to improving the quality of the GENERIS system.

6 Conclusions and discussion

This thesis identified the mandatory responsibilities of a Finnish DSO and multiple business processes were identified based on the responsibilities. Most significant of the processes are those related to registering the measurements and communicating them to relevant parties. Another important process is the nation-wide balance settlement lead by the imbalance settlement responsible party which is Fingrid in Finland. The processes were formalized as BPMN diagrams. This was done to support the test designing.

A test case for market messaging was designed and implemented using the QM framework. The requirements for the test were formalized using metric temporal logic and propositional logic. The test case detected errors as was expected. Therefore it can be concluded that QM framework is eligible for testing market messaging processes and it can be assumed that the framework is applicable to any process that utilizes the public interfaces of GENERIS.

Time shifting also seems to be a working concept as the test ran successfully for more than 4 months of virtual time. The reason for not reaching a full year was not an error in the test implementation but a genuine error in the SUT itself. The finding was reported and analyzing this should contribute to the quality of the GENERIS system.

It is very difficult to rigorously classify the implemented test case because it has traits from multiple methodologies. Another thing that complicates the classification is the fact that the boundary between runtime verification and oracle based testing is obscure. The implemented test approach is thus best described as a combination of black-box system testing and runtime verification.

If the SUT is working correctly, the approach is closer to runtime verification as the test continues for a whole virtual year. However, if the SUT produces an incorrect output, the test stops and is regarded as a failure. In traditional black-box testing, a test is executed completely and the verdict is done based on the output.

A number of requirements for the runtime verification monitors were formalized. There are frameworks that allow creating the monitors directly from formal requirements, an example of such is utilized by Artho et al. [2005]. However, the QM framework doesn't support creating the monitors directly from the requirements, and the requirement logics must be implemented in C# as part of the test case. Creating monitors automatically from MTL statements is a possible topic for future research.

Even though QM framework is an eligible tool for testing the GENERIS MDM system it is not without flaws. Currently the QM test development is extremely time consuming and complex compared to AutoTester. Additionally, since the test scripts are pure C# code the development requires a lot of programming experience. Otherwise a lot of time will be consumed by familiarizing oneself with the common C# paradigms. Inexperience also leads to more time consumed by troubleshooting

the test case itself.

AutoTester has a clear advantage over QM in this regard. The scripts are very straightforward and are easy to develop even without prior knowledge of the AutoTester. For example, a reference test case that was developed in this thesis took only three hours to develop and debug without prior AutoTester experience. The development and debugging of the QM test took 50 hours with no prior knowledge about QM framework.

The testing approach affects the difference in development times as well. The QM has more functionalities that were implemented, such as input data generation, MSCONS message reading logic and the validation based on the data extracted from the MSCONS message. AutoTester doesn't have any of these features so this naturally affects the time consumption and complexity. The QM test could also utilize preconstructed inputs and outputs but that is not desired since it doesn't make sense to build an identical system with AutoTester.

Due to its nature, AutoTester is more suitable for low level functional testing. In this case, low level functionality means a functionality of an integrated GENERIS system, rather than source code level functionality. AutoTester can directly execute IFM jobs and access data in the GENERIS' database which makes it suitable for this type of testing. The difficulty, however, is determining the expected outputs as the the scripts are static and functionalities can be time dependent.

From the experiences and evidence gained from this thesis it is clear that while QM framework can be used for testing it is still far from complete. A suggestion to improve the test development process would be to adopt a keyword-based layer on top of the QM framework. In this scenario, a test designer would use the keyword-based layer to design the test and the actual test developer would develop the corresponding functionalities in the underlying QM framework. This way, the test designers would be free to design better tests since they would be relieved from the actual coding work. The developed functionalities could then be used in the keyword layer in the later tests as well.

Another approach would be to integrate the QM framework and AutoTester system together in order to take advantage of the best sides of both of them. The QM framework could be used to host the test cases and test steps and to generate the input and possibly the expected data. The steps could then be implemented as simple AutoTester scripts.

The oracle can be within QM or AutoTester depending on the implementation. If the expected output data would be generated as well, the implementation would resemble the normal automatic test oracle. Additionally, QM could modify the AutoTester script's constant variables according to input parameters before execution which would lessen the need for AutoTester script maintenance.

AutoTester supports command line execution so the technology for this kind of integration exists already since a C# program can execute command line commands.

The keyword-based solution, however, would support the test designing process better so that should be preferred. Other researches also support keyword-based test automation, as was discussed in section 3.

All in all, future research should be directed towards making QM test development more cost efficient. Another topic that should be addressed in future research is the feasibility of third party testing tools and frameworks compared to the QM framework. The investments in QM framework development should be compared to the costs of commercially available solutions in order to determine, whether developing and maintaining an in-house framework is reasonable.

The relevance of this thesis is mostly limited to Enoro company and GENERIS system. An internally developed testing framework was applied to company's own software. However, the first research objective, analysis of the most important processes, applies to MDM systems in Finland generally. Even though the processes were derived from GENERIS point of view, the responsibilities are common. From Enoro's point of view, the most important aspect of this thesis is the QM feasibility analysis. The analysis helps in the planning of the future of the testing.

7 Summary

Quality management is defined as the actions that are taken to direct the quality of a product to a desired state. Quality means the product's ability to meet the expressed and implied functional and non-functional requirements. Quality management consists of different sub-activities one of which is quality assurance.

Software quality assurance process is defined in IEEE 730 standard. The process can be divided into three subsets of activities: SQA process implementation activities, product assurance activities and process assurance activities. The product assurance activities are the actual actions that provide evidence about the quality of the software product. SQA implementation activities include planning, coordinating and documenting the SQA process. The process assurance activities check that the processes comply with the regulations and are able to provide quality products.

Product assurance activities use different software verification methods to provide confidence in the quality of the software. Software testing is a common way to find evidence that the product meets the requirements. Testing can be done on multiple levels in the software development and delivery process. Additionally, multiple verification methods exist. The methods can be classified to static, dynamic and exhaustive methods.

Dynamic testing methods find software defects by executing the program. These methods can be further classified to black-box, white-box, and gray-box testing based on the information that is available to the test. Black-box testing means that the source code is not available to the test. Typically, the system level testing is black-box testing. The QM framework is intended solely for system level black-box testing.

Runtime verification is a software verification approach where the correctness of the software is evaluated during an actual execution of the system. Runtime verification can also be considered as oracle based software testing. The testing approach that was implemented in this thesis is best described as a combination of black-box system testing and runtime verification. In case of a successful test, the approach resembles runtime approach more closely. Multiple temporal and non-temporal requirements were identified and the monitors were implemented as part of the test case.

This thesis focused on the quality assurance for an MDM system called GENERIS. An MDM system governs all the metering data in a smart grid. The MDM system stores and manages the data for billing and balance settlement purposes. This thesis identified the most important processes of GENERIS based on Finnish laws and regulations. The most important identified processes were balance calculations and the related messaging and market messaging to retailers. Another notable processes are estimation of missing measurements and converting readings to hourly measured powers.

Quality Manager is a new testing framework developed within Enoro that features

time shifting functionality. The main goal of this thesis was to analyze the feasibility of the QM framework and the time shifting in the testing of the critical processes. The analysis was conducted by implementing a time shifting test for the market messaging process. It was concluded that time shifting and the framework can be utilized in the market messaging testing. A test execution was run successfully for over four months of virtual time. The reason for not reaching a full year was finding an actual error in the SUT.

In addition, the QM framework and the implemented test was compared with an existing automatic test execution system called AutoTester. AutoTester scripts proved to be significantly simpler and faster to develop. However, in terms of script maintenance the QM prevails over AutoTester. AutoTester doesn't have any internal data structures, so it can't be used to generate the input data or the expected output. These have to be preconstructed and they are often time dependent.

It was concluded that the QM framework is not yet ready. QM test development requires too much resources as it is. However, it shows potential and it was suggested that another key-word based layer could be implemented on top of the framework so that the test development would be easier. Another option to investigate is the combination of QM framework and AutoTester. In this scenario, the QM framework would be used for input data generation and to make the verdict about the correctness of the observed behaviour.

The effects on a standard SQA process were also analyzed. The QM framework affects the SQA implementation activities heavily since the SQA planning must be reviewed in order to utilize the new framework effectively. In addition, the utilization must be carefully coordinated with other processes and verification tools in order to minimize the redundancy. The product assurance activities are also affected heavily if a new framework is adopted to the verification toolbox.

All in all the thesis was a success as all the research targets were addressed. The business process analysis helps to design tests for the new framework. The framework was found out to be usable but it requires further development to be effective. The implemented test is valuable, as it was shown to detect errors in the software that have previously passed the SQA process unnoticed.

References

F. Shull, V. Basili, B. Boehm, A.W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz. What we have learned about fighting defects. In *Proceedings Eighth IEEE Symposium on Software Metrics*, volume 0, pages 249–258. IEEE Comput. Soc, 2002. ISBN 0-7695-1339-5. doi: 10.1109/METRIC.2002.1011343. URL `http://doi.ieeecomputersociety.org/10.1109/METRIC.2002.1011343http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1011343`.

Brent Hailpern and Padmanabhan Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12, 2002.

Vahid Garousi and Junji Zhi. A survey of software testing practices in Canada. *Journal of Systems and Software*, 86(5):1354–1376, may 2013. ISSN 01641212. doi: 10.1016/j.jss.2012.12.051. URL `http://dx.doi.org/10.1016/j.jss.2012.12.051`.

Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, K. Petersen, and M. V. Mäntylä. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *Automation of Software Test (AST), 2012 7th International Workshop on*, pages 36–42, June 2012. doi: 10.1109/IWAST.2012.6228988.

Suomen Standardoimisliitto SFS. Quality management systems - fundamentals and vocabulary. *SFS-EN ISO 9000:2015*, 2015.

Dongping Tang and Junshan Chen. Exploring the Impact of Software Quality Management on Informatization. *2010 International Conference on E-Business and E-Government*, (70832003):2700–2703, 2010. doi: 10.1109/ICEE.2010.682. URL `http://www.scopus.com/inward/record.url?eid=2-s2.0-78649649156&partnerID=tZOtx3y1`.

Umair Sajid Hashmi, Naveed Anjum, and Aqeel Israr. Impact of Software Quality Standards on Commercial Product Development and Customer Satisfaction for Software Industry in Pakistan. In *2013 Fifth International Conference on Computational Intelligence, Modelling and Simulation*, pages 269–274. IEEE, sep 2013. ISBN 978-0-7695-5155-5. doi: 10.1109/CIMSim.2013.50. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6663196`.

IEEE. Ieee standard adoption of iso/iec 90003:2014, software engineering – guidelines for the application of iso 9001:2008 to computer software. *IEEE Std 90003-2015*, pages 1–71, Sept 2015. doi: 10.1109/IEEESTD.2015.7274039.

IEEE. Ieee standard for software quality assurance processes. *IEEE Std 730-2014 (Revision of IEEE Std 730-2002)*, pages 1–138, June 2014. doi: 10.1109/IEEESTD.2014.6835311.

G. Hongying and Y. Cheng. A customizable agile software quality assurance model. In *Information Science and Service Science (NISS), 2011 5th International Conference on New Trends in*, volume 2, pages 382–387, Oct 2011.

Shahela Saif, Aliya Ashraf Khan, and Fahim Arif. An analysis of a comprehensive planning framework for customizing sqa. In *Proceedings of the 2010 National Software Engineering Conference*, NSEC '10, pages 2:1–2:7, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0026-1. doi: 10.1145/1890810.1890812. URL http://doi.acm.org/10.1145/1890810.1890812.

Jae Won Lee, Sung Hwa Jung, Sung Chang Park, Young Joong Lee, and Young Chul Jang. System based SQA and implementation of SPI for successful projects. In *IRI -2005 IEEE International Conference on Information Reuse and Integration, Conf, 2005.*, volume 2005, pages 494–499. IEEE, 2005. ISBN 0-7803-9093-8. doi: 10.1109/IRI-05.2005.1506522. URL http://www.scopus.com/inward/record.url?eid=2-s2.0-33745728500&partnerID=tZOtx3y1http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1506522.

IEEE. Ieee standard for system and software verification and validation. *IEEE Std 1012-2012 (Revision of IEEE Std 1012-2004)*, pages 1–223, May 2012. doi: 10.1109/IEEESTD.2012.6204026.

Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, may 2009. ISSN 15678326. doi: 10.1016/j.jlap.2008.08.004. URL http://dx.doi.org/10.1016/j.jlap.2008.08.004.

ISO/IEC/IEEE. Systems and software engineering – vocabulary. *ISO/IEC/IEEE 24765:2010(E)*, pages 1–418, Dec 2010. doi: 10.1109/IEEESTD.2010.5733835.

Tim a Majchrzak. *Improving Software Testing*. SpringerBriefs in Information Systems. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-27463-3. doi: 10.1007/978-3-642-27464-0. URL http://link.springer.com/10.1007/978-3-642-27464-0.

Ilene Burnstein. *Practical software testing : a process-oriented approach*. Springer, 2003. ISBN 9786610188451.

Mumtaz Ahmad Khan and Mohd Sadiq. Analysis of black box software testing techniques: A case study. In *The 2011 International Conference and Workshop on Current Trends in Information Technology (CTIT 11)*, number i, pages 1–5. IEEE, oct 2011. ISBN 978-1-4673-0098-8. doi: 10.1109/CTIT.2011.6107931. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6107931.

ISO/IEC/IEEE. Iso/iec/ieee international standard for software and systems engineering–software testing–part 4: Test techniques. *ISO/IEC/IEEE 29119-4:2015*, pages 1–149, Dec 2015. doi: 10.1109/IEEESTD.2015.7346375.

Tom Wissink and Carlos Amaro. Successful Test Automation for Software Maintenance. In *2006 22nd IEEE International Conference on Software Maintenance*, pages 265–266. IEEE, sep 2006. ISBN 0-7695-2354-4. doi: 10.1109/ICSM.2006.63. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4021345.

S. Berner, R. Weber, and R.K. Keller. Observations and lessons learned from automated testing. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 571–579. IEEe, 2005. ISBN 1-59593-963-2. doi: 10.1109/ICSE.2005.1553603. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1553603.

Christer Persson and N. Yilmazturk. Establishment of automated regression testing at abb: industrial experience report on 'avoiding the pitfalls'. In *Proceedings. 19th International Conference on Automated Software Engineering, 2004.*, pages 112–121. IEEE, 2004. ISBN 0-7695-2131-2. doi: 10.1109/ASE.2004.1342729. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1342729.

Nan Li and Jeff Offutt. An Empirical Analysis of Test Oracle Strategies for Model-Based Testing. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 363–372. IEEE, mar 2014. ISBN 978-1-4799-2255-0. doi: 10.1109/ICST.2014.49. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6823898.

Vineeta, Abhishek Singhal, and Abhay Bansal. A study of various automated test oracle methods. In *2014 5th International Conference - Confluence The Next Generation Information Technology Summit (Confluence)*, pages 753–760. IEEE, sep 2014. ISBN 978-1-4799-4236-7. doi: 10.1109/CONFLUENCE.2014.6949222. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6949222.

Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. A comprehensive survey of trends in oracles for software testing. *University of Sheffield, Department of Computer Science, Tech. Rep. CS-13-01*, 2013.

Yliès Falcone and Lenore D. Zuck. Runtime verification: the application perspective. *International Journal on Software Tools for Technology Transfer*, 17(2):121–123, apr 2015. ISSN 1433-2779. doi: 10.1007/s10009-014-0360-z. URL http://link.springer.com/10.1007/s10009-014-0360-z.

Séverine Colin and Leonardo Mariani. *18 Run-Time Verification*, pages 525–555. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. ISBN 978-3-540-32037-1. doi: 10.1007/11498490_24. URL http://dx.doi.org/10.1007/11498490_24.

Rajeev Alur and T.A. Henzinger. Real-time logics: complexity and expressiveness. In *[1990] Proceedings. Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 390–401. IEEE Comput. Soc. Press, 1990. ISBN 0-8186-2073-0.

doi: 10.1109/LICS.1990.113764. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=113764.

Prasanna Thati and Grigore Roşu. Monitoring Algorithms for Metric Temporal Logic Specifications. *Electronic Notes in Theoretical Computer Science*, 113(SPEC. ISS.):145–162, jan 2005. ISSN 15710661. doi: 10.1016/j.entcs.2004.01.029. URL http://linkinghub.elsevier.com/retrieve/pii/S1571066104052570.

N. Delgado, A. Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering*, 30 (12):859–872, Dec 2004. ISSN 0098-5589. doi: 10.1109/TSE.2004.91.

Energiateollisuus. Energian mittaus, 2016. URL http://energia.fi/sahkomarkkinat/sahkoverkko/energian-mittaus. Accessed: 5.2.2016.

V. Cagri Gungor, Dilan Sahin, Taskin Kocak, Salih Ergut, Concettina Buccella, Carlo Cecati, and Gerhard P. Hancke. A Survey on Smart Grid Potential Applications and Communication Requirements. *IEEE Transactions on Industrial Informatics*, 9(1):28–42, feb 2013. ISSN 1551-3203. doi: 10.1109/TII.2012.2218253. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6298960.

Fingrid, 2016. URL http://www.fingrid.fi/fi/asiakkaat/Tiedonvaihtopalvelut/rekisteripalvelu/Sivut/default.aspx. Accessed 31.7.2016.

D. Matheson, Chaoying Jing, and F. Monforte. Meter data management for the electricity market. In *Probabilistic Methods Applied to Power Systems, 2004 International Conference on*, pages 118–122, Sept 2004.

M. Jung, W. Kastner, G. Kienesberger, and M. Leithner. A comparison of web service technologies for smart meter data exchange. In *2012 3rd IEEE PES Innovative Smart Grid Technologies Europe (ISGT Europe)*, pages 1–8, Oct 2012. doi: 10.1109/ISGTEurope.2012.6465825.

Työ- ja Elinkeinoministeriö. Sähkömarkkinalaki 588/2013, 2013. URL http://www.finlex.fi/fi/laki/alkup/2013/20130588. Accessed 23.8.2016.

Fingrid. Ediel sanomavälityksen yleiset sovellusohjeet, 2015. URL http://www.fingrid.fi/fi/asiakkaat/asiakasliitteet/Tiedonvaihto/2015liitteet/Edielsanomav{%}C3{%}A4lityksenyleisetsovellusohjeet.pdf. Accessed: 21.4.2016.

M Tariq, Z Zhou, J Wu, M Macuha, and T Sato. Smart grid standards for home and building automation. In *2012 IEEE International Conference on Power System Technology (POWERCON)*, pages 1–6. IEEE, oct 2012. ISBN 978-1-4673-2868-5. doi: 10.1109/PowerCon.2012.6401448. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6401448.

eSett. Nordic imbalance settlement handbook, 2016. URL http://www.esett.com/wp-content/uploads/2016/03/NBS-Handbook-v2.13.pdf. Accessed 1.9.2016.

Ediel. Message handbook for Ediel - Functional Description, 2002. URL https://www.ediel.fi/sites/default/files/MessageHandbookforEdielFunctionalDescriptionversio2.4A.pdf. Accessed: 16.7.2016.

Ediel. Message handbook for Ediel - Implementation guide for METERES SERVICES CONSUMPTION REPORT. 2005. URL https://www.ediel.fi/sites/default/files/ImplementationGuideforMSCONSversio2.4D.pdf.

Energiateollisuus Ry. Sähkömarkkinoiden käytännön menettelyohje III, 2013. URL http://energia.fi/sites/default/files/images/sahkomarkkinoiden_kaytannon_menettelyohje_iii_20130314.pdf.

Vahid Garousi and Michael Felderer. Developing, Verifying, and Maintaining High-Quality Automated Test Scripts. *IEEE Software*, 33(3):68–75, 2016. ISSN 0740-7459. doi: 10.1109/MS.2016.30. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7412621.

Earl T Barr, Mark Harman, Phil Mcminn, Muzammil Shahbaz, and Shin Yoo. The Oracle Problem in Software Testing : A Survey. *IEEE Transaction On Software Engineering*, 41(5):507–525, 2015. ISSN 0098-5589. doi: 10.1109/TSE.2014.2372785.

Yuhong Zhao and Franz Rammig. Model-based Runtime Verification Framework. *Electronic Notes in Theoretical Computer Science*, 253(1):179–193, 2009. ISSN 15710661. doi: 10.1016/j.entcs.2009.09.035. URL http://dx.doi.org/10.1016/j.entcs.2009.09.035.

Stephen H. Edwards. A framework for practical, automated black-box testing of component-based software. *Software Testing, Verification and Reliability*, 11 (2):97–111, jun 2001. ISSN 0960-0833. doi: 10.1002/stvr.224. URL http://doi.wiley.com/10.1002/stvr.224.

Eun Ha Kim, Jong Chae Na, and Seok Moon Ryoo. Implementing an Effective Test Automation Framework. In *2009 33rd Annual IEEE International Computer Software and Applications Conference*, volume 2, pages 534–538. IEEE, 2009. ISBN 978-0-7695-3726-9. doi: 10.1109/COMPSAC.2009.188. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5254082.

Tuomas Pajunen, Tommi Takala, and Mika Katara. Model-Based Testing with a General Purpose Keyword-Driven Test Automation Framework. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 242–251. IEEE, mar 2011. ISBN 978-1-4577-0019-4. doi: 10.1109/ICSTW.2011.39. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5954415.

Alex Cervantes. Exploring the use of a test automation framework. In *2009 IEEE Aerospace conference*, pages 1–9. IEEE, mar 2009. ISBN 978-1-4244-2621-8. doi: 10.1109/AERO.2009.4839695. URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=4839695&contentType=Conference+Publications&queryText=Exploring+the+use+of+a+test+automation+framework.

Työ- ja Elinkeinoministeriö. Työ- ja elinkeinoministeriön asetus sähköntoimitusten selvitykseen liittyvästä tiedonvaihdosta 809/2008, 2008. URL http://www.finlex.fi/fi/laki/alkup/2008/20080809. Accessed 23.8.2016.

Markku Rissanen, Jari Mustaparta, Janne Pirttimäki, Jarmo Roiha, Juuso Ruottinen, Saku Ruottinen, Joel Seppälä, Aarne Sievi, Riina Heinimäki, and Elina Lehtomäki. Tuntimittauksen periaatteita, 2010. URL http://energia.fi/sites/default/files/tuntimittaussuositus_2010_paivitetty_20121204.pdf.

M. Buckley and R. Chillarege. Discovering relationships between service and customer satisfaction. In *Software Maintenance, 1995. Proceedings., International Conference on*, pages 192–201, Oct 1995. doi: 10.1109/ICSM.1995.526541.

Cyrille Artho, Howard Barringer, Allen Goldberg, Klaus Havelund, Sarfraz Khurshid, Mike Lowry, Corina Pasareanu, Grigore Roşu, Koushik Sen, Willem Visser, and Rich Washington. Combining test case generation and runtime verification. *Theoretical Computer Science*, 336(2-3):209–234, may 2005. ISSN 03043975. doi: 10.1016/j.tcs.2004.11.007. URL http://linkinghub.elsevier.com/retrieve/pii/S0304397504007753.

A Case example error report

Distributing messages the sender and receiver data goes wrong and message is sent to wrong party Created: 14.04.2016 Updated: 11.05.2016 Resolved: 18.04.2016	
Status:	Closed
Project:	
Component/s:	None
Fix Version/s:	None

Type:	Bug	**Priority:**	Normal
Reporter:		**Assignee:**	
Resolution:	Fixed		
Labels:	None		
Remaining Estimate:	0h		
Time Spent:	7h		
Original Estimate:	0h		
Environment:			

Criticality:	Blocker
Deadline:	15.04.2016
Deadline Info:	It is in prod and affects FG
Backlog:	CS 3rd Level Backlog
Branch To Fix:	2.11 R1, Mainline
Definition of Done:	☑ ~~Implementation done~~ ☑ ~~Testing passed~~ ☑ ~~Documentation done~~
Test Status:	Passed
Documentation:	<Describe here how the issue was documented>
Checked into branch:	GENERIS Main, V2.11 R1
Released In Version:	2.11.2.0, 2.11.1.10
Release Notes:	Sorting GXML elements caused problems with saxbasic-scripts that assume certain order inside GXML. GXML sort disabled for distributions.
Participants:	
Last Resolution User:	

Instructions:	14.04.2016 1. Open distribution model: "MSCONS_tuntimitattavat_YYY" 2. Execute manual distribution for period 14.01.2016 - 16.01.2016 3. Check the message in SPool - MSCONS OUT. After created the file, the Tunniste is not normal. It should contain the retailer code where to the message will be sent. **XXX**:160:SLY_XXX_3900393 <-- WRONG **YYY**:160:SLY_YYY:SLY:EON_5447917 <-- RIGHT 4. To compare the messages, we can provide some other test systems where test is in 2.11.1 and prod is in 2.9.1 and same distribution model is executed.
Surround Events:	Click to see Surround events related to issue

Description

Created GXML-message fields for receiver and sender goes crossed or something and message is sent to wrong party after edi-file creation.

Comments

Comment by J [14.04.2016 - Visible by: pv-users]

is this only 2.11.1? FG is 2.9.1

Comment by H [14.04.2016 - Visible by: pv-users]

Tämä johtuu kahdesta asiasta
1. MSCONS_export_MSG_SPOOL_NDC.bas on seuraavanlainen koodi eli jos xml sisältää yhden ”recipient ” elementin, niin otetaan ensimmäinen ja jos niitä on enemmän niin otetaan jälkimmäinen. Tässä siis ei tutkita tarkemmin kumpi recipienteistä mahtaa olla lähettäjä ja kumpi vastaanottaja.

```
Set recipNodes = objCollNode.selectNodes("rec:recipient")
With recipNodes
Set sendNode=.Item(0)
If .length < 2 Then
Set recipNode=.Item(0)
Else
Set recipNode=.Item(1)
End If
End With
```

Valitulla recipient elementillä on edielId, joka määrää viestin tunnuksen ja jota käytetään sitten lähetystietona.

```
<rec:recipient id="13" key="aonkojioqllapqkkepqbeecqdnjococllknginjmhoqimp">
<rec:channelType enumid="1">FILE</rec:channelType>
```

<rec:edielId>YYY:SLY:EON</rec:edielId>
<rec:internalID>19AA7C41-0208-421E-A5F3-B93A2CE1F874</rec:internalID>
<rec:reportMethod enumid="2">EDIEL</rec:reportMethod>
<rec:notifyChgOutBal>true</rec:notifyChgOutBal>
</rec:recipient>

<rec:recipient id="7" key="aonkojioqllapqkkepqbeecqdnjococllknginjmhoqhjgoh">
<rec:channelType enumid="2">FTP</rec:channelType>
<rec:edielId>XXX</rec:edielId>
<rec:internalID>E288C405-5B5A-46CB-BFEB-87E102B13336</rec:internalID>
<rec:reportMethod enumid="2">EDIEL</rec:reportMethod>
<rec:notifyChgOutBal>true</rec:notifyChgOutBal>
</rec:recipient>

2. Nyt uusimmassa GENERIS versiossa nämä recipient-objektit voivat olla missä järjestyksessä tahansa, koska GXML export järjestää elementit tyypin mukaan.

Ongelma voidaan hoitaa skriptiä muuttamalla kunhan tiedetään kumpi noista recipienteistä on vastaanottaja.

Mutta koodiakin voisi muuttaa sen verran, että jakelumalleissa otetaan sorttaus pois päältä.

Comment by H [14.04.2016 - Visible by: pv-users]

MSCONS_export_MSG_SPOOL_NDC.bas file fixed to mainline and 2.11.1 branches and testserver EDIEL folder.

Comment by A [15.04.2016 - Visible by: pv-users]

Approved in XXX production that fix works OK and Aperak is received frim retailers.

Comment by M [18.04.2016 - Visible by: pv-users]

Sorting GXML elements caused problems with saxbasic-scripts that assume certain order inside GXML.
GXML sort disabled for distributions.

Comment by J [19.04.2016 - Visible by: external-users]

JMU:n ollessa työmatkalla ohjattu issue Klle.
-JTA

Comment by J [19.04.2016 - Visible by: pv-users]

väärä issue, tama takaisin ADD:lle.

Comment by J [20.04.2016 - Visible by: pv-users]

Tätä ei suljeta ennen kuin ADD on saanut tarkistettua tähän tehdyt korjaukset.

Comment by A [20.04.2016 - Visible by: pv-users]

2.11.1.10 beta 1 Test OK XXX FAT
2.9.2.39 beta 1 Test OK NNN FAT